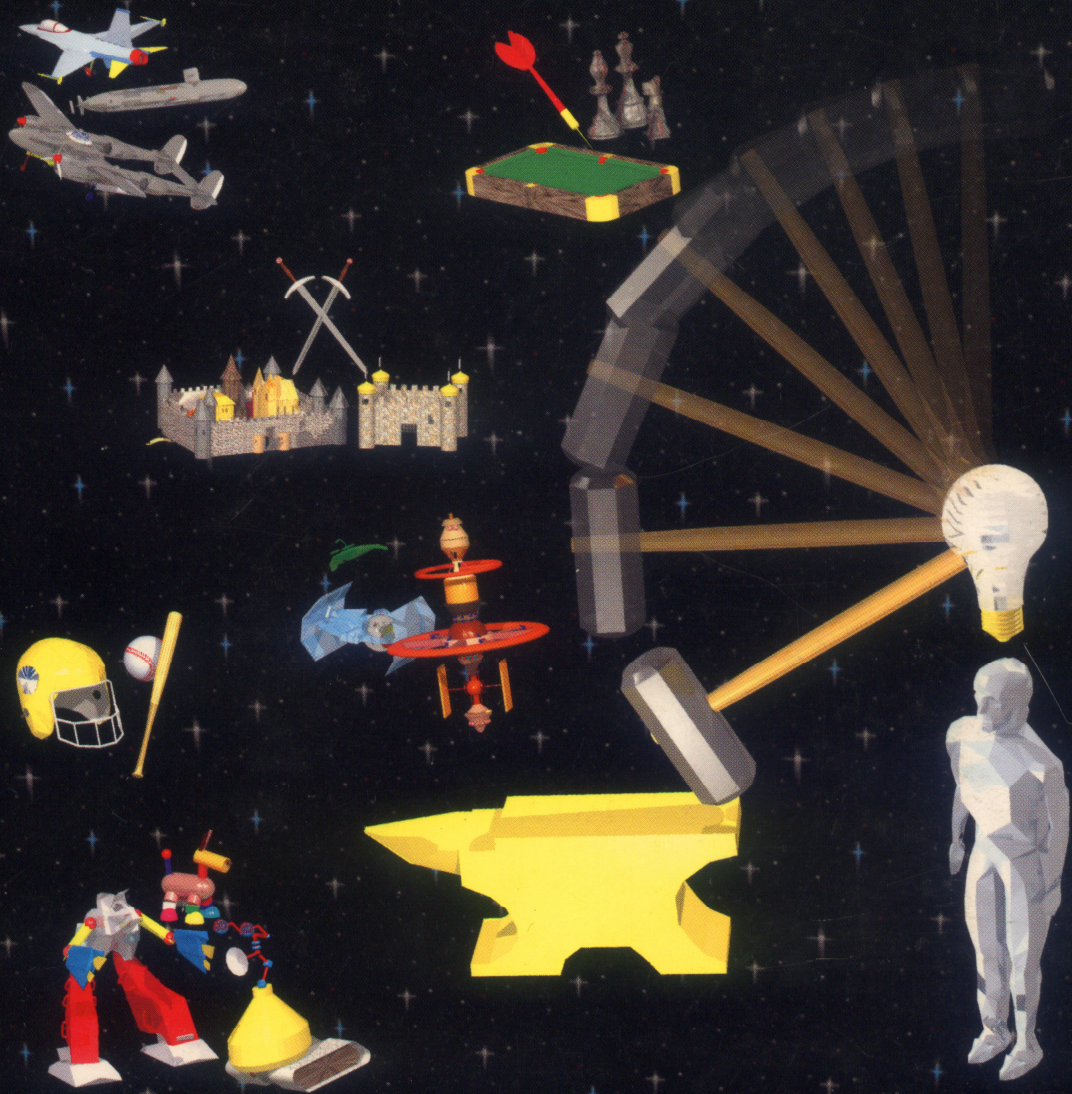


GAMESMITH

Development System



Professional Game Development
for your Amiga Computer

Innovative software from



GAMESMITH

Development System

Reference Manual

Professional Game Development
for your Amiga

Innovative Software From

OREGON



RESEARCH

Copyright Notifications

Oregon Research

16200 S.W. Pacific Highway
Suite 162
Tigard, OR 97224

Customer Support: (503) 620-4919
FAX: (503) 624-2940
Internet: orres@teleport.com
Genie: ORA
CompuServe: 71333,2655

GameSmith is a trademark of Bithead Technologies.
Amiga Installer and Workbench Copyright © Commodore-Amiga
Example programs are copyright by their respective authors.

Program Copyright ©1994 Bithead Technologies. All rights reserved.
Manual Copyright ©1994 Bithead Technologies and Oregon Research.
All rights reserved.

Published and distributed exclusively by Oregon Research. Neither the program nor this manual may be reproduced in any form without express written permission from Oregon Research.

Program Design: John Enright
Programming: John Enright
Manual by: John Enright, Bob Luneski, and Jo Luneski

A special message from the program author:

Thanks go to God for giving me a talent.
To my parents for always being there to support me.
For Mountain Dew that kept me awake, and Miller that let me relax.
To the United States of America, for this wonderful opportunity.
May we never loose our faith.

-John Michael Enright

Software License Agreement

THE INSTALLATION OF "THE GAMESMITH DEVELOPMENT SYSTEM", HEREINAFTER REFERRED TO AS "THE SOFTWARE", INDICATES YOUR EXPLICIT AGREEMENT TO THE TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE AGREEMENT.

OREGON RESEARCH GRANTS YOU A NONEXCLUSIVE LICENSE TO INSTALL AND USE THE SOFTWARE ON A SINGLE MACHINE DESIGNATED BY YOU. BACKUP COPIES OF THE SOFTWARE MAY BE MADE FOR ARCHIVAL PURPOSES ONLY. AT NO TIME SHALL MORE THAN ONE COPY OF THE PROGRAM BE IN USE SIMULTANEOUSLY. RENTAL OF THIS SOFTWARE IS EXPRESSLY PROHIBITED BY THIS LICENSE.

OREGON RESEARCH MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. REGARDING THE SOFTWARE, OREGON RESEARCH DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND THE PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

IN NO EVENT WILL OREGON RESEARCH OR IT'S AGENTS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR INDIRECT DAMAGES(INCLUDING DAMAGES FOR THE LOSS OF BUSINESS PROFITS, LOSS OF BUSINESS INFORMATION AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE EVEN IF OREGON RESEARCH HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

OREGON RESEARCH'S LIABILITY TO YOU FOR ACTUAL DAMAGES FROM ANY CAUSE WHATSOEVER, AND REGARDLESS OF THE FORM OF THE ACTION (WHETHER IN CONTRACT, TORT(INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE), WILL BE LIMITED TO \$50.

Table of Contents

Copyright Notification	ii
Software License Agreement	iii
1 Getting Started	1
Introduction	1
The GameSmith Development System	2
Installation	3
Technical Support	4
Acknowledgment	6
Mastering GameSmith	6
2 GameSmith Tutorial	8
Designing the Game	8
Defining the Parts	8
Overview of GameSmith Systems Used	10
Using CITAS to Create Anim Objects	11
Object-to-Object Collision Detection	12
Object-to-Background Collision Detection	15
Sound	16
The Game Program	16
Putting it all together	18
3 The Graphics System	19
The Picture Loader	19
The IoadILBM Structure	19
The BitMapHeader Structure	22
The Bitmap Structure	22
The GameSmith Color Table	23
Blitter Operation	23
CPU Blittable Objects	26
User Copper Lists	26
Graphics Functions	30
4 The Display System	31
Double Buffering	31
The Viewport Structure	32
The Display Structure	35
Multiple Viewports in a Single Display	38
Scrolling	38
AGA Mode Considerations	39
Dual Playfield	40
Parallax Viewports	42
Sliced Parallax Viewport	44
Display Functions	47
5 The Animation System	48
Structure Hierarchy	49
Bitmap Handling and Display Priority	49

Object Types	50
Object Save Areas	51
Virtual Space and Real Space	52
Bounds Checking	53
Collision Detection	53
Object-to-Object Collision Detection	53
Object-to-Background Collision Detection	54
Display Methods	55
Attached Anims	56
Disk Based Anim Objects	57
Linked Anim Objects	57
Object Arrays	58
Objects in Dual Playfield Mode	58
The Background Collision Structure	59
The Object Collision Structure	60
The Anim Structure	61
The Anim Complex Structure	65
The Anim Load Structure	66
Hardware Sprites	67
Attached Sprites	68
Animation Functions	68

6 CITAS - Object Animation Tool 71

Image Creation	71
Output Types	71
CITAS Objects & the Animation System	72
Display Modes	72
System Preferences	73
Creating a Simple Anim Object	74
Image Colors and Anim Colors	77
Image Alignment	77
Anim Options	78
Animation Preview	80
Creating a Complex Anim Object	81
Anim Files	83
Anim Complex Files	84
Anim Header Files	84
Collision Tables	86
Collision Masks	86
Object-to-Object Collision	88
Object-to-Background Collision	92
Source Output	93
Source Code Labels	93
Additional Header Files	94
Keyboard Controls	95

7 The Sound System 96

Introduction	96
Operation	96
The Sound Structure	97
Sound Functions	99

8 Utility Functions	100
The AmigaDOS Librarian	100
CYDEC Encryption Module	100
Joystick Polling	102
Vector Routines	102
Random Number Generator	103
Task Priority	103
Audio Filter Control	103
File Requestor	103
Vertical Blank Timer	104
Vertical Blank Functions	104
Plot Routines	104
Bitmap Allocation	104
Utility Functions	104
9 Using The GameSmith Library	105
Linking With The GameSmith Library	105
Using Locked Anim Files	106
Using Encrypted Graphics and Sound Files	106
Calling Library Functions From 'C'	106
Calling Library Functions From Assembler	107
SAS C 6.5 Users Note	108
Understanding the Library Reference	108
10 Library Quick Reference	110
Graphics Functions	110
Display Functions	110
Animation Functions	111
Sound Functions	113
Utility Functions	113
11 GameSmith Library Reference	114
Addendum	189
General Index	191
Library Function Index - By System	197
Library Function Index - Alphabetical	199

1 Getting Started

Introduction

The GameSmith Development System (GDS) is a powerful game development package that has evolved through years of research and careful design. With it, you will find all of the parts necessary to build everything from Street Fighter II to Lemmings to Dungeon Master - and beyond! This system can be thought of as providing the chassis, transmission, wheels, and engine of a powerful sports car. All you have to do is bolt it together, give it a paint job, and you'll be driving a hot racer. Spark that with a little ingenuity, and you'll be building robots and space stations as well!

Any type of video game is programmable using GDS. From shoot'em ups to graphics adventures, from intergalactic conquest to strategic simulations. We provide the low level muscle and you provide the creative gaming skills. Finally, the power of professional, arcade quality video game programming in a single easy to use system.

The GameSmith library functions are provided in the form of a linker library. This means that the user is not restricted by any given programming environment (like BASIC), but allows the programmer to harness the power of the Amiga with the best development languages available. The only restriction is that it supports linking standard Amiga object files.

The default language supported by GameSmith is 'C' and all of the examples and functions will be defined in terms of C syntax. Assembly language is also a very good choice for game development. Where there are important differences for Assembly Language programmers, such as when parameters must be placed on the stack etc., these differences are noted.

In addition to the powerful elements of the system, the programmer also has the revolutionary animation utility CITAS. There has always been a simple problem facing any game developer, the solution of which has been daunting to all but the most hardcore developers. How do I get my graphics into my game? And beyond that, how do I align animations and detect when they bump into things?

The solution in the past, was to use any of a number of crude utilities to turn an IFF ILBM image file into 'C' or assembler source code. The source code could then be compiled or assembled and then referenced from a program as graphics data. The other common solution involved simply creating raw, unformatted graphic files.

Unfortunately, most of these solutions are a little like crude oil: they have to be refined somehow to make them usable. Traditionally, games were simply hard coded to fit the graphics, and a lot of special attention was necessary to make sure everything fitted together just right. This very rigid formula means rewriting code every time the graphics change. A very tedious and time consuming process.

CITAS changes the way game designers will deal with graphics. CITAS can use IFF ILBM graphics from any paint program. In addition to its ability to generate source code in 'C' or assembler, it includes all of the complex controlling structures and data that make up anim objects in the powerful GameSmith animation system. And not just for a single image, but any number of images to combine into any number of separate, distinct animation sequences. All combined into a single graphics object!

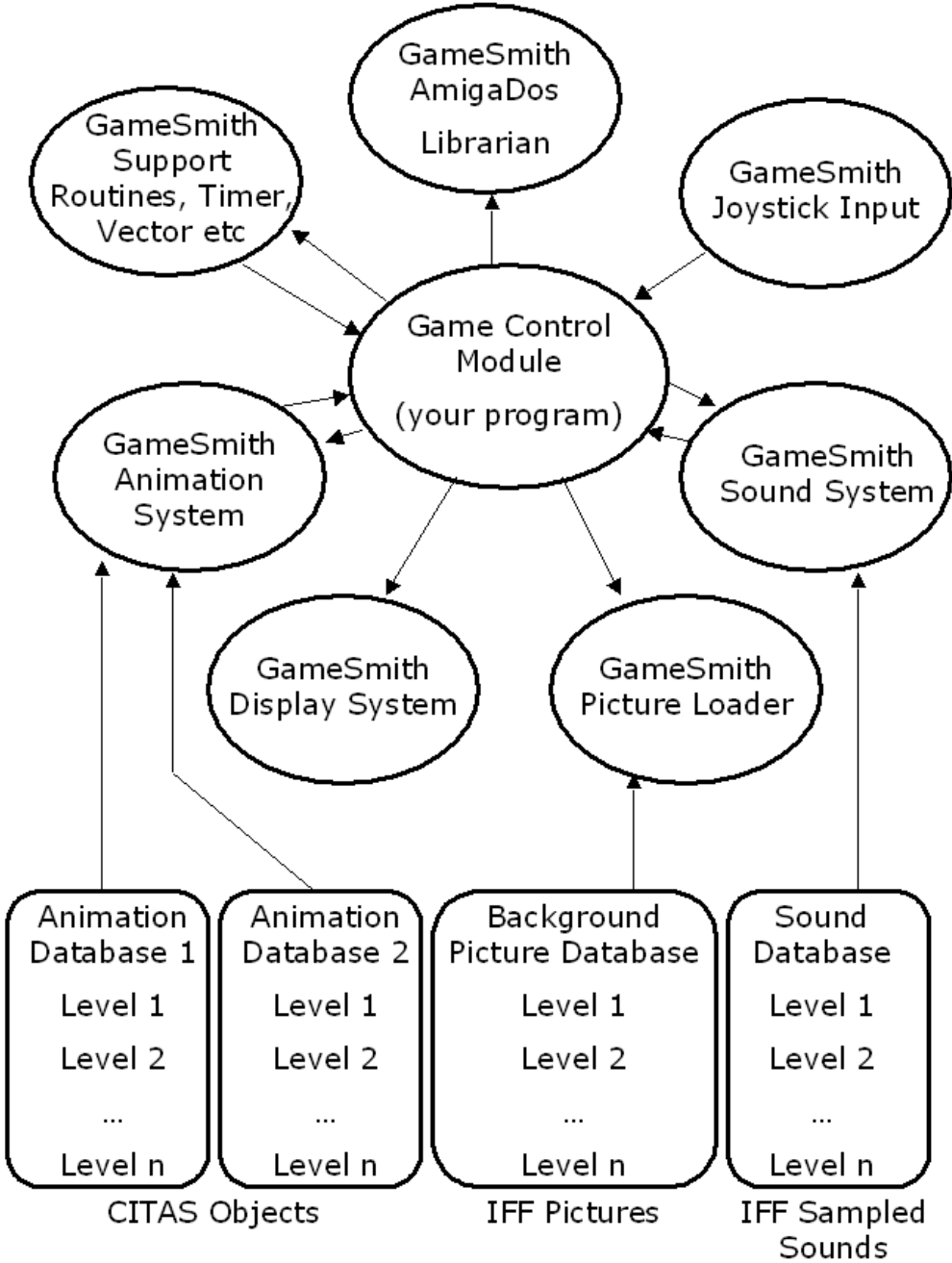
Beyond the ability to create source from animations, CITAS can write binary animation files that can be simply loaded into your programs with a single line of code. Through its simple graphical user interface the user can easily build all of these things, and preview his or her work. Even more, CITAS allows the user to graphically specify object-to-object and object-to-background collision detection areas for his or her animations, assign logical names, and decide which collision types can collide with each other.

All of this information is embedded in the objects created by CITAS. Your program can find out everything it wants to know about an anim object from the object itself. This means no need for hard coding and brings true object oriented programming to graphic objects, and a new level in game design.

OK, at this point you're probably saying, "But, how do I use it?" The following chapters describe, in detail, each of the major parts of the GameSmith Development System. Here you will find a layout of all of the controlling structures, a full explanation of the fields within those structures, and "the big picture" on how things work including a complete tutorial walking you through the creation of an actual game.

You can see from the chart on the this page that each GameSmith system module operates completely separate from the other modules; neither relying on, nor communicating with other modules. Any or all of the GameSmith modules may be included in or excluded from your program as you see fit, depending on the library functions which your program calls.

The GameSmith Development System

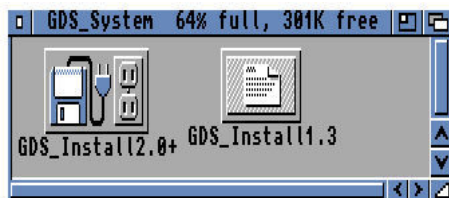


Components of the GameSmith Development System

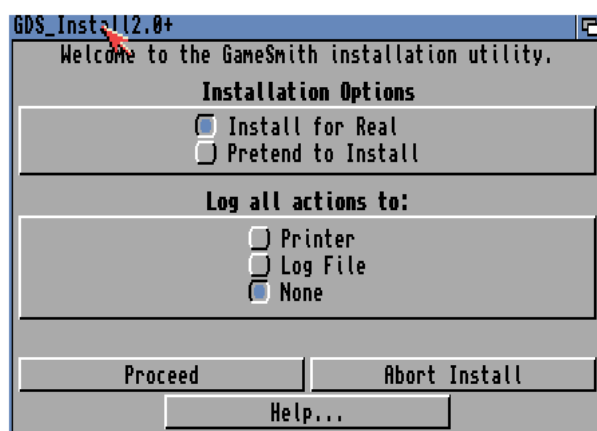
Installation

A hard drive, although not required, is highly recommended for use with the system. To install GameSmith Development System follow the steps listed below:

- Place your GameSmith Program disk in DF0:
- Double click on the GameSmith drive icon to open a window



- You will see a window like the one shown above. Now, doubleclick on the GDS_Install Icon to launch the installation program.



- Click on "Proceed" to begin the installation process.
- Follow the on screen prompts to complete the installation process.

It is important to note that ALL users, including floppy only users, must complete the installation procedure. Each GameSmith Development System is uniquely serialized. The installation process validates your serial number and embeds key information in GameSmith system files that allows you the ability to lock your anim files, making them usable only by you and the games you develop. Your serial number can be found on your Program disk. Please take the time to write it down in the box on the next page.

After you have completed the installation, remember to fill out and mail in your product registration card. Registration is EXTREMELY important! It is required to receive technical support. In addition, we need to know how to get a hold of you to keep you informed of product upgrades and enhancements.

Technical Support

Oregon Research's technical support staff is trained to provide you with fast and courteous service. Your purchase of the GameSmith Development System comes with 90 days of free technical support via letter, phone, FAX, and electronic mail. After 90 days free technical support is still available, but only via postal letter. Please return your completed owners registration card to become eligible for technical support.

If you require technical support via phone, FAX, or electronic mail beyond the 90 day free support period, extended technical support plans are available for a nominal charge. Contact Oregon Research in North America or HiSoft in Europe for more information on extended support plans.

Please be aware that we cannot teach you 'C' or Assembly language programming. Technical support is limited to the implementation of the GameSmith Development System itself.

If you require assistance beyond what the manual, help features, and README files can provide, then call or write to us with the information listed in the technical support checklist below. Before calling, please attempt to recreate the problem so that you can provide us with an exact sequence of events. If the problem reoccurs, then contact us by phone, FAX, or electronically on GENIE or CompuServe with the following information:

- **Product Information:** The name, version number, and file date/time of the application program. The version number appears in the About box.
- **Your user registration number:** This appears on the master disk label. Write it in the box provided NOW!

Serial Number:	0495-200-120386
----------------	-----------------

- **System Information:** You'll need to know, the Amiga computer model that you own, including memory configuration and operating system version.
- **Additional Hardware:** You'll need to know the brand names of any additional hardware that you have connected to your system. This includes disk drives, printers, modems, accelerators, custom video cards, PC emulators etc.
- **Startup Sequence and Resident programs:** You will need a list of All programs run from your startup sequence or that were resident at the time of the problem. You should first try eliminating unneeded programs to see if a conflict exists.
- **Error Messages:** Write down the EXACT wording of any error messages that you received from the program.
- **Attempted solutions:** Make careful note of any steps you took to solve the problem and what the results were.

If you need assistance beyond these suggestions, and you cannot find the answer to your problem in the manual, help features, or README files, then please write, call, or FAX us with the information in the technical support checklist. We need all of the information in the checklist to efficiently serve you.

In North America contact:

Oregon Research
16200 S.W. Pacific Hwy., Suite 162
Tigard, OR 97224
USA

Phone: (503) 620-4919

FAX: (503) 624-2940

Electronic Support:

Genie: ORA
CompuServe: 71333,2655
Internet: orres@teleport.com

In Europe contact:

HiSoft
The Old School
Greenfield, Bedford
UNITED KINGDOM MK45 5DE

Phone: +44 525 718181 FAX: +44 525 713716

Electronic Support:

Internet: hisoft@cix.compulink.co.uk

Acknowledgment

The linker libraries of the GameSmith Development System are provided free from any royalty or acknowledgment requirements. However, as a professional courtesy, Oregon Research requests that you give some mention of the GameSmith Development System in your products or product packaging. Favorable mention of this package will help guarantee additional support and enhancements in the future. We at Oregon Research would like to thank you for purchasing this product. We are committed to increasing the quality and abundance of game development tools.

Mastering GameSmith

Before you go off on a tour of GDS, remember to protect your investment! Make a copy of the GameSmith floppy disks, and store the originals in a safe place.

Everyone has their own way of learning a new program. However, the GameSmith Development System is not just any program. It is an extremely deep and rich development environment. Do not expect to be writing games like the Shadow of the Beast or Lemmings in your first week with the system. It will take some time to learn all of the power and capabilities of the system.

With that in mind, we have a recommended method for learning the system. You can, of course, follow whatever method you are comfortable with. However, we guarantee that following the steps we suggest, in the order that we suggest them WILL decrease your learning curve. This means by investing a little time up front, you will be writing the game of your dreams better and faster than otherwise.

How to Master GameSmith in 10 easy steps:

- Study the GameSmith System Components chart on page 2 again. Pay special attention to the system relationships. Always remember who is at the center controlling everything (Look in the mirror for a hint :-).
- Read the Table of Contents to become familiar with the major sections of the manual.
- Read the tutorial in Chapter 2. Don't worry too much about details, you're just trying to get a feel for the system.
- Do not read, but skim the Chapters 3-10. Pay more attention to concepts than to specific details or implementation. This will give you a basic understanding to the fundamental concepts of the system without getting overwhelmed by the quantity of information.
- Quickly browse Chapter 11, the Library Reference, to get familiar with the names of some of the available functions.
- Run CITAS and play around with the menus. Again concentrate on just getting familiar with some of the options.
- Now, go back and read chapters 3-10. However, this time, carefully read the sections and spend more time trying to understand a few more of the details.
- Now, you're ready to get your feet wet! It's time to write your first game. CAREFULLY read the tutorial Chapter 2. But this time, don't just read the tutorial. DO THE TUTORIAL. Carefully study the code. Write the game along with it, use CITAS to edit the animations, do everything involved with the tutorial. Even though the code is there for you do it yourself too.
- This tutorial has been very carefully designed to exercise most of the major components of the system without excessive complexity. **There is no better way to learn the GDS than spending a large amount of time studying the tutorial and the balance of the example code.**

- After you feel you have mastered the tutorial, it's time to dive into the rest of the example and demo code. Each demo program exercises different aspects of the system. Again, there is no better way than doing. Study the code carefully. Understand the calling sequence to accomplish what the demo accomplishes. Then change it! Make it do something new. Make it YOURS!
- Spread your wings and fly. Once you feel you have mastered the basics of the system. Write your first game. Try to keep the complexity level down on your first effort. Ease and comfort with the system will come quickly, but be careful not to attempt too ambitious a project too soon.
- **Have Fun! And remember that there is no better way to learn the GDS than spending a large amount of time studying the tutorial and the balance of the example code.**

2 GameSmith Tutorial

The GameSmith Development System is a rich and powerful programming environment. Probably the best way to learn the system is by actually using it. This tutorial has been carefully designed to demonstrate the fundamental concepts and components of the system. Do not be concerned with the kind of game chosen for the tutorial, the concepts learned are applicable to any kind of game. Concentrate on the concepts, functions, and techniques being demonstrated.

We recommend that you go through the tutorial fairly quickly the first time. This will give you a basic familiarity of the system. Then carefully study the rest of the manual. After you have done that, return to the tutorial and go through it again. This time carefully study the code paying special attention to the concepts and techniques being demonstrated.

Then get creative. Play with it, modify the code, add new features, make it yours! There is no better way to learn the system than to use it in an actual game. Spending a lot of time carefully studying this tutorial will significantly increase your knowledge of the system and speed the development of your first game.

Designing the Game

The game we'll be making for this tutorial will be a tank battle game. It's a simple game, but demonstrates many common elements found in arcade style shoot'em-ups. It has been carefully designed to demonstrate the fundamental concepts and elements of the system.

Before we can begin, we must first DESIGN the game. This first step is a critical part of good programming practices. Our first design step is to define the actual rules of the game itself. So:

- This will be a two player game, requiring two players each to control his/her single tank with a joystick.
- The object of the game will be simply to shoot the other tank, and be the first player to reach the winning score.
- The winning score will be 21, and the player must win by 2 points over his/her opponent (i.e. if you reach 21, but your opponent has 20, then game play continues until one player has a score that is 2 more than the other player).
- Shooting your own tank, via a bouncing bullet, will cause you to lose a point from your score, with a minimum score of 0.
- Game play will take place on a non-scrollable battlefield, with player scores presented at the bottom of the screen. The battlefield itself will contain obstacles which neither the tanks nor their bullets can pass through.
- The game will have an options screen. Hitting the Esc key during game play will return to the options screen. The game will exit cleanly back to the Workbench screen when the "Quit" option is selected.

If you've ever played the Atari 2600 tank battle game, you'll have a pretty good idea of what we're building, only our version will have better graphics and sound.

Defining the Parts

Now that we've defined the rules, we can make a list of the various pieces we will need for this game, and how those pieces will interact:

- Two tanks, one for each player. We will use DPaint and CITAS to create simple anim objects out of the tanks. Each tank will have a distinctive color so that players can tell the tanks apart.
- Two bullets. Each tank may have one bullet in the air at any time. The bullets will be animated, so we will be creating a complex animation consisting of an animation sequence for each orientation of the tank images (12 different positions in all). You will be shown how this is easily accomplished with CITAS.
- Two explosion animations, one for each bullet. These will be simple anim objects, invisible until a bullet impacts on something, and becoming invisible again after the last frame of the explosion has been displayed.
- Battlefield backdrop picture (DPaint IV will be used to draw the battlefield, as with all graphics used in this tutorial, though any paint program which can save in IFF ILBM format will suffice). This picture should have walls and other obstacles.
- Score and options. For this we will use an Amiga RastPort structure and the system Move() and Text() functions to display our text. This is the easiest way to draw words on our GameSmith display, and the speed is adequate. For this tutorial, we'll use the system supplied ROM font, TOPAZ 8, though you are free to design and use your own font.

We will present the players with some simple options:

- Bullet Speed (1 to 4).
- Bullet Bounce (yes or no)
- Battlefield [future implementation for choosing battlefield]
- New Game
- Quit
- Resume Game [only displayed for an ongoing game]

Bullet speed will be multiplied times the tank movement rate to determine the actual speed of the bullet in pixels. Enabling bullet bounce will allow bullets to ricochet off of obstacles and the edges of the battlefield when struck at an angle. Selection of different battlefields (option c above) may be left as an exercise for the reader.

- Sound samples. We will need:
 - a sound for tank movement.
 - a sound for when a shot is fired.
 - a sound for when a bullet explodes on an obstacle.
 - a sound for when a bullet explodes on a tank (a score).
 - a sound when player 1's bullet bounces off of an obstacle.
 - a sound when player 2's bullet bounces off of an obstacle.
- Collision detection. Bullet and Tank objects will interact in the following manner:
 - bullet → background obstacles
 - bullet ↔ bullet
 - bullet ↔ tank
 - bullet → edge of the display bitmap
 - tank → background obstacles

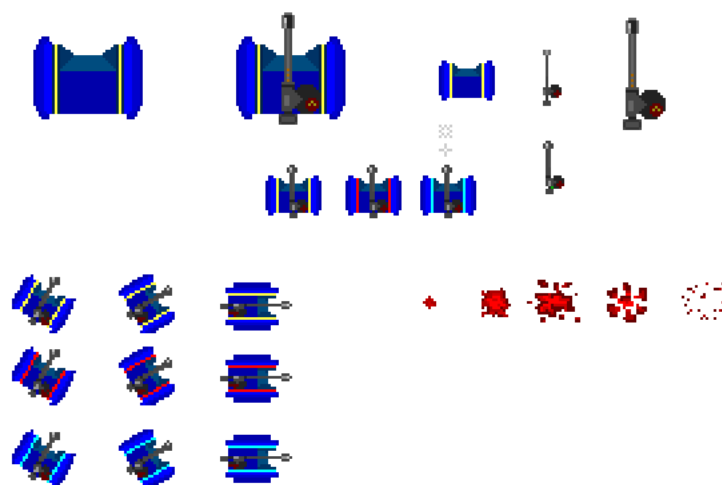
Overview of GameSmith Systems Used

- A GameSmith display with 2 viewports. The top viewport will show the battlefield, while a small bottom strip displays scores. The game will try to open a PAL display (256 lines vertically), so NTSC users with ECS and AGA machines should make sure that the PAL monitor device is installed in their devs:monitors drawer. The game does not require PAL, however, and will automatically adjust the display size for non-ECS/AGA NTSC users, or NTSC users running AmigaDOS 1.3 or below.
A low resolution, non-interlaced, six bitplane extra halfbrite display was chosen for this game. However, on stock 68000 machines, the added display bandwidth required for 6 bitplanes vs. 5 bitplanes (64 color vs. 32 color) may slow down the computer too much for full frame rate operation. For this reason, a 32 color version of the game is also included. The program was simply recompiled to use 5 bitplanes instead of 6, and without the EXTRAHALFBRITE display mode.
- The animation system will be used heavily. We will also be loading our anim objects (tank, bullet, and explosion) from disk. The anim objects will remain freely editable by CITAS (not final output versions).
- The sound system. We will be loading IFF 8SVX sound samples from disk. The samples included with this tutorial will not be encrypted, and are public domain samples.
- The IFF ILBM picture loader for loading the battlefield backdrop and setting the color palette. Battlefields will also not be encrypted so that the user is free to modify them or draw their own.
- Blitter functions for clearing and copying a bitmap.
- Miscellaneous utility functions such as the random number generator and the vertical blank timer.

Graphics

Now that we've got the game basics defined, we'll need some graphics. Pulling out a handy dandy paint program (DPaint IV in this case), the first thing we'll need to do is define our color palette. You'll find that this is a critical and necessary step before any graphics can actually be drawn. The color palette (or palettes in some cases) should be chosen carefully, and should generally be given much consideration.

An error in the color palette may mean redrawing some or all of your graphics. Once the color palette is selected, we can set about drawing the tank, bullet, and explosion images.



Since CITAS provides an image rotation tool (only in 90 degree increments at the time of this writing), you needn't save a brush for every angle of the tank image (12 in all). In this case, DPaint was used to rotate the original image 30 degrees, 60 degrees, and 90 degrees. The 90 degree image rotation was a little disproportionate due to differing horizontal to vertical pixel aspects, and so required a little touch up.

Once all images are created, clip them out individually and save them to disk as brushes. Remember that you don't have to clip images to exact pixel edges, because CITAS automatically removes blank space around images. The graphic images were drawn on a green background because that's how the game will look. However, you'll need to first change the green to the background color (color 0) with a fill option before clipping and saving your image brushes.

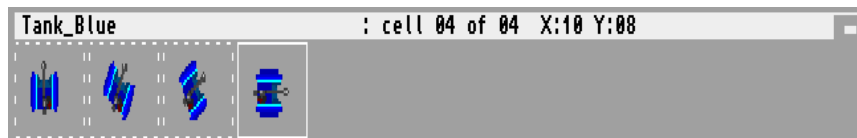
Using CITAS to Create Anim Objects

It's now time to start building our animated objects. Run the interactive graphics animation tool CITAS by double clicking on it's icon or running it from the CLI.

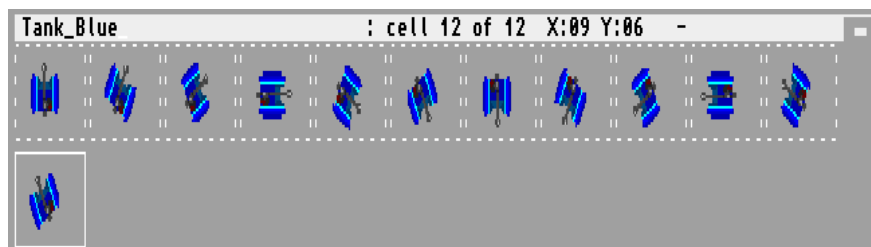
The Tanks

It will be easiest to start by building the tank objects, so we'll begin here. Both gold and blue tanks will be created in the same manner. Let's define the blue banded tank image with 0 degrees rotation (gun turret facing upwards) as the first image, and load that one. Call the newly created anim object Tank_Blue (CITAS will prompt you for a name after you load the 1st image). Let's then load the images so that when animated the object will rotate in a clockwise motion. This was chosen arbitrarily, and could just have easily rotated in a counter clockwise motion within CITAS. The game program must be aware of how the object was built, however, so that it can coordinate animation with joystick movement.

Once all 4 tank brushes have been loaded your CITAS display should look like this:



Now, we'll need to generate the remaining 8 tank positions. Copy cell 2, and rotate that image 90 degrees using the menu options for those functions. Copy cell 3, and rotate that image 90 degrees. Copy cell 1, and rotate that image 180 degrees (two rotate operations). You now have 7 of the 12 possible tank positions. In the same manner, copy and rotate the images until you have 12 separate frames as shown below.

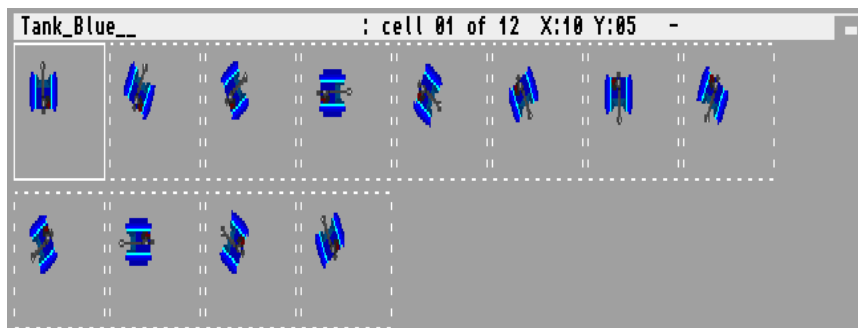


Because the tank images are not symmetrical from left tread to right tread, the rotate operation is necessary instead of a flip to maintain image consistency.

Now we'll need a little room to work with as we align the images into a smooth animation, so hit the ctrl-right_arrow twice and the ctrl-down_arrow once. This increases the size of the box around each image, but this does NOT require any additional memory on the part of the object. It merely allows us to define different placement offsets for the object images. We will now have enough room, after moving the images toward the center of each cell, to allow the bullet to show within each frame (we'll get to that later).

Start animation preview by hitting the up arrow key. You'll notice that the tank doesn't seem to rotate around its center. We're going to fix that, so hit the space bar to pause the animation. Single step through the frames with the left or right arrows until you get to frame 1. Move this image towards the center of the frame. An X offset of 10, and a Y offset of 5 works well. Flip between frames 1 and 2 with the left and right arrow keys, and shift image 2 until it looks correct when rapidly flipping between the two images. Repeat this process until all images are aligned properly.

(Now, continue animation preview by hitting the up arrow again to check your work. You can speed up animation or slow it down with the up and down arrows. Your display should now look like the one shown below. At this point, save your work. It's always good to save work in progress periodically.



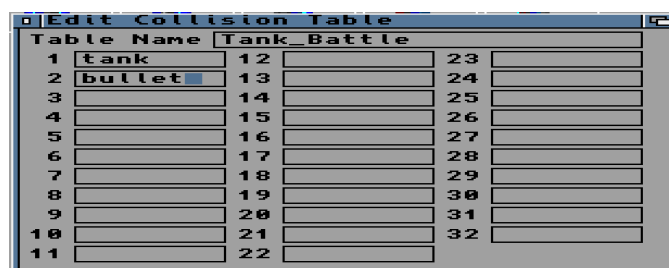
Object-to-Object Collision Detection

Before we go any further, now is a good time to define the collision table we'll use in this game. A collision table is used for object-to-object collision detection. In this case, we'll need to detect when a bullet hits a tank or another bullet. Select the "Collision Tables" option from the Anim menu.

This will bring up a list of collision tables currently in memory. The list will be empty. In the text entry box, type in "Tank_Battle", and click on the NEW button.



You've now created a collision table, but it's empty. Click on the EDIT button to define the area types for this table. The tab and return keys will move between fields. For this simple game, we only need two types of collision areas: a tank area, and a bullet area. Let's name them "tank" and "bullet" respectively. When finished, click on the close window gadget. All changes are kept.

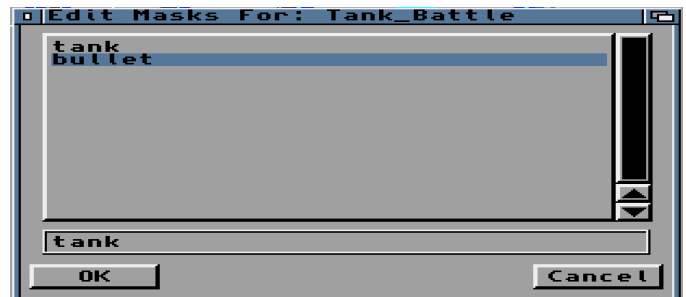


Now that we have actual area types in the table, we'll need to define which areas can collide with each other. Click on the MASKS button from the collision tables list. Each defined area in the table will allow you to define a mask. In other words, you specify what other areas each area can collide with.

The first area in the list is presented in the rectangular box below the list (in this case "tank"). To allow tank areas to collide with other areas, click on their names in the list. This will highlight them. Selecting a highlighted area name (clicking on it) will disable collisions for that area (the item will no longer be highlighted).

For this game, we'll need tank areas to collide with bullet areas, but not with other tank areas, so just highlight "bullet". Click on the OK button to move to the next area name in the list (in this case "bullet").

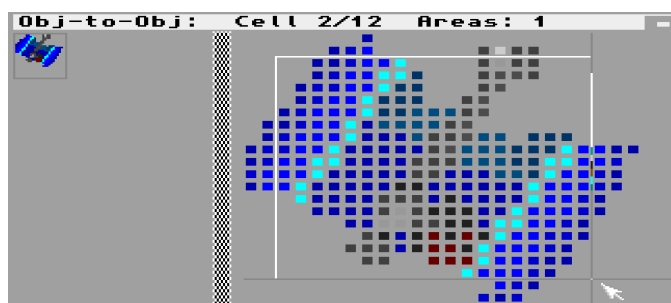
We'll need the bullet areas to be able to collide with both the tank areas and other bullet areas, so highlight them both in the list and click on OK.



We've just created a collision table and defined how our collision areas can collide with each other. Now the animation system will call our collision handler when two collision enabled areas overlap each other. Now, close the collision tables list window.

All that remains now is to draw the actual collision areas on each image of our anim objects. First we need to tell CITAS to associate a collision table with an anim object, so now select "Select Collision Table" from the Anim menu.

A list of collision tables in memory will be displayed. Double click on Tank_Battle. From here, select Collision Areas >> Object-to-Object from the Image menu. This changes the display to the zoom window mode (see the CITAS manual for details), and prompts you to select an area type. It defaults to the first entry in the list, which is "tank". This is the area type we want for our tank anim, so just click on OK.



Now simply draw a single box around each image. Use the left and right arrows to move to the previous or next image in the anim. The menu bar will show you which anim cell you're currently working with. When completed, return to the main menu and save the anim object to disk. Repeat this same process for the gold tank images. Believe it or not, this is the hard part of making this game (well, this and drawing the graphics).

The Bullets

You might be wondering, after playing the game, how the bullet objects always look like they're being shot out of the tank's gun turret, no matter which direction the tank is facing. There's a little trick for doing this, but it involves making an anim object for all 12 directions. It's actually quite easy, after you get the hang of it. We'll be combining all 12 bullet anims into an anim complex, and so our bullet objects in the game are actually complex anim objects (complete details on anim complexes are presented later in the manual).

OK, let's start building the first bullet anim. Make sure the Tank_Blue anim is loaded, and is the currently displayed anim on the CITAS screen. Select Rename Anim from the Anim menu, and give it the name "bullet1". Now delete every cell in the anim except the first one. Next, load the 2 bullet images (the brush files saved earlier). Align the bullet images (both will use the same X and Y offsets) so that when rapidly animated, the bullet images lie directly in front of the tank's gun turret as shown below.



When you're satisfied with the bullet image placement, delete the first image (the tank image). Now we're almost done (easy, isn't it). Now because we just renamed the bullet1 anim from Tank_Blue, it is already using the Tank_Battle collision table. Now, go to the object-to-object collision screen, select the bullet area type, and draw a little box around both bullet images. Return to the main menu.

There's only one little detail remaining: object priority. Since we want the bullet object to appear on top of the tank object if they overlap, we need to assign a higher display priority to each bullet object. Select Options from the Anim menu, and change the default display priority of 0 (lowest) to something higher. Leave the other options at their defaults.

That's it, we've completed one bullet anim. Save this object. Next, reload the Tank_Blue anim and repeat this process until you have 12 bullet anims in memory (use the 2nd tank cell for bullet2, and so on for all 12 tank images). The basic process is:

- Reload Tank_Blue
- Rename the anim
- Delete all images except the one you want to work with
- Load both bullet brushes
- Align bullet images in front of gun turret
- Create "bullet" type collision areas over bullet images
- Increase the display priority (make all bullet anims the same)

Now that we have all 12 bullet anims in memory, let's combine them into a single anim complex. Select List Complexes from the Anim menu. The list will be empty, since we haven't created one yet. Type in "bullet" and click on the NEW button. Now click on ANIMS.

We're going to add all 12 bullet anims to the bullet complex, but the way in which we do it is very important. Add each bullet anim, from the 1st to the 12th, IN ORDER FROM LOWEST TO HIGHEST (i.e. from 1-12). This will make sure that the bullet anim sequences correspond to the tank anim images.

See the Chapter 6, CITAS, if you have trouble adding anims to an anim complex. Save the anim complex to disk (select Save Complex from the Project menu).

The Explosion

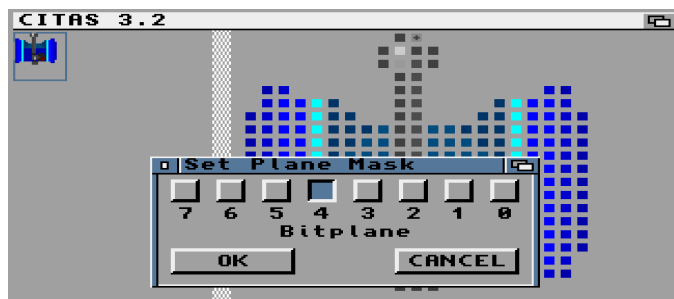
The explosion anim is very easy to build, and we don't need to define any collision detection for it. Simply load, sequence, and align the explosion images in the same way you loaded the tank anim and save the anim to disk.

Object-to-Background Collision Detection

Only one thing remains and our anim objects for this game will be finished. We need to have the tank and bullet objects interact with the background data. The background color values we'll be concerned with, and how objects will react to them are as follows:

- Colors 16, 28, and 29 (of 0 - 63) will be significant to our game.
- Color 28 will be considered a vertical wall.
- Color 29 will be considered a horizontal wall.
- A tank cannot pass through any of these color values.
- Bullets will explode on color 16, and bounce off of 28 and 29 (if bullet bounce is enabled, otherwise bullets will blow up). Bullets will bounce in the appropriate manner based on the color value causing the collision.

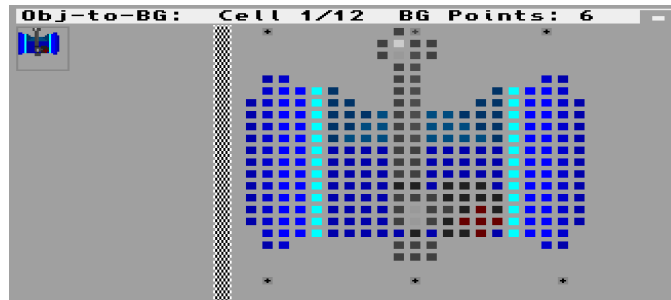
So let's set up the tank objects first. Reload the tank objects into CITAS and select the Collision Areas >> Object-to-BG option from the Image menu. Since all of the colors we'll be concerned with use the 5th bitplane (for colors 16 - 31), we can set up our collision mask so that only the 5th plane causes an object-to-background collision. Deselect all other plane indicators by clicking on them. Only plane 4 should be depressed. This way we can avoid needless calls to our background collision handler by the animation system.



Now that we've defined the collision mask for background imagery, we need to be only slightly careful with the tank setup. Our tanks can roll forward or backward, depending on which direction the player is pushing the joystick. The game program will need to know which end of the tank (the front or the back) is bumping into something when a collision occurs.

Within the `coll_bg_struct` (the structure created by CITAS for each object-to-background collision detection point) there is a field called "area". This field contains the area number, from 0 to n, of each point. The area number is assigned starting with 0, and incrementing through the total number of points for any given image. For instance, if we put 6 points on each tank image (and we will), their numbers will be 0 - 5, and will be assigned in the order in which we define the points. Therefore, if we place the first 3 collision points in the front of the tank, and the last 3 behind the tank, our program can merely check if the area number is less than 3 to determine if the collision has occurred in the front of the tank.

Place 3 collision points across the front of each tank image, and 3 across the back IN THAT ORDER. Once that's done, save the tank object, and do the other tank in the same way. Now the tanks are done.



The bullet anims are much easier than the tanks. Use the same collision mask (only plane 4 collision enabled), but simply place a single point in the center of each bullet image. Do this for all 12 bullet anims, and resave the bullet complex.

Now we're done with the anim objects. Very little remains before we code the actual game.

Sound

The GameSmith sound system has been carefully designed to allow easy playback of sampled sounds. We will be loading and playing several public domain sound samples in conjunction with various events during game play.

The samples themselves are in IFF 8SVX format, so we can rest assured that the GameSmith sound system will play them back exactly as they should sound. While shopping for sound samples for this tutorial, the little SND utility (found on your example disks) came in very handy.

Good sound sampling hardware and accompanying editing software are indispensable tools when you need to create your own sounds. This was necessary for most of the smaller sounds in this tutorial. An excellent choice for creating and editing very high quality 8 bit sampled sounds direct to disk is MegaloSound. For higher quality sounds and more advanced editing software, the 16 bit sampler Clarity 16, and the new 12 bit direct to disk sampler, Aura, are excellent high quality, low cost solutions to all your sampling requirements. These products are available directly from Oregon Research and MicroDeal or from your local Amiga dealer.

The Game Program

With all of the pieces in place, it's time to write the actual game. Because of the popularity of the 'C' language, this tutorial will be written entirely in 'C'. Using the GameSmith Development System, it should be possible to write any sort of game in 'C'.

The following is a pseudo-code layout for the game program. Pseudo-code is a very useful technique that is like a strategic plan. It's not actual code, but rather the interrelationships and functions of the program modules. It is essentially a hand written flow chart for the program. Note that the pseudo-code does not do any error handling; it's intent merely to lay out a working model. Refer to the actual 'C' source code for details on actual game implementation.

main

```
Call setup present game options while (NOT quit option) { call battle if (user abort) present game options else if (winning score) { congratulate winner present game options } } call cleanup end
```

setup

Load color palette from battlefield picture create double buffered display load battlefield picture into display load anim objects load sounds initialize animation system and add objects install obj-to-obj collision handler install obj-to-bg collision handler return

options

Display options while (NOT quit, new game, or resume) { get user input relate user input to game variables (bullet speed, bounce, etc.) } if (new game) initialize game variables (tank positions, score, etc.) return

battle

For (each player) { if (explosion active) animate bullet explosion, remove once complete if (tank hit) remove bullet, start explosion, and update scores if (bullet active) move and animate bullet if (bullet hit edge of bitmap) { if (bullet bounce is YES) call bounce else remove bullet and start explosion anim } get joystick input if (LEFT) turn tank counter-clockwise if (RIGHT) turn tank clockwise if (UP and NOT bumping into anything in front of tank) move tank forward if (DOWN and NOT bumping into anything behind tank) move tank backward if (FIRE button and bullet NOT active) activate bullet and select correct anim for tank position } clear bump indicators for front and back of tanks return

obj-to-obj collision

If (bullet hit tank) { flag to remove bullet flag tank hit if (tank shot itself) if score is greater than 0, subtract 1 from score else add 1 to score } else bullet hit bullet, so activate explosions and flag to remove bullets return

obj-to-bg collision

If (tank object) { if (color is equal to 16, 28, or 29) { if (collision area is < 3) flag front of tank bumping into something else flag back of tank bumping into something } } else bullet object { if (color is equal to 16) flag to remove bullet and start explosion if (color is equal to 28 or 29) { if (bullet bounce is equal to YES) call bounce else flag to remove bullet and start explosion } } return

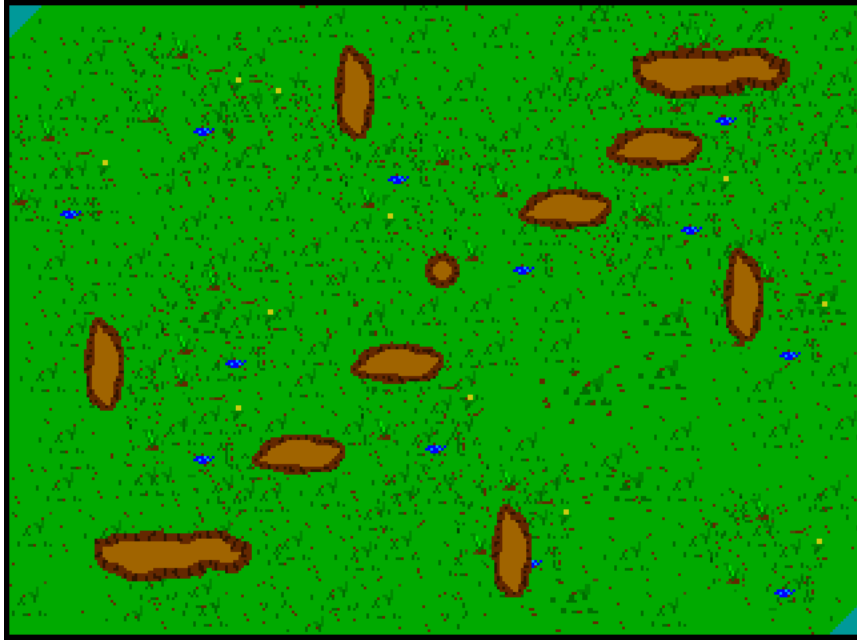
Bounce

if (bullet moving straight left, right, up, or down) flag to remove bullet and start explosion else if colliding with vertical wall bounce to the left or right of wall at opposite angle of impact else colliding with horizontal wall bounce to the top or bottom of wall at opposite angle of impact return

cleanup

free anim objects free sounds free display return

Putting it all together



After you have a basic understanding of the pseudo-code, carefully examine the actual source which can be found on your Examples disks. Pay particular attention to how strategies defined by the pseudo-code are actually implemented in the GameSmith Development System. Once you're comfortable, have some fun and make some changes. Get creative, remember the only limit is your imagination!

3 The Graphics System

This chapter describes the fundamentals of the GameSmith Graphics System, and the basic principles behind their operation.

The Picture Loader

The GameSmith picture loader is an easy yet versatile tool for loading standard Amiga IFF ILBM pictures from disk. Pictures are loaded with a single call, and placed into one or two supplied bitmaps. In addition, the picture loader may optionally allocate the bitmap(s) for you, ensuring that they are large enough to contain the specified picture file. The picture loader can also optionally fill a color table for you so that you have access to the picture's color palette. Experienced programmers can even cause the loader to shuffle or depth arrange image bitplanes.

The GameSmith ILBM loader has the added bonus of being able to read files that were encrypted with the `gs_cypher` program (see the section on encryption in Chapter 8 Utility Functions for more information). This powerful capability allows you worry free distribution of your graphics.

The loadILBM Structure

The IFFILBM picture loader requires a properly initialized load structure.

```
struct loadILBM_struct {
    unsigned char *file;
    struct BitMap *bitmap1;
    struct BitMap *bitmap2;
    unsigned long *color_table;
    int num_colors;
    int dheight;
    int dwidth;
    int xoff,
    int yoff;
    int modes;
    int loadx;
    int loady;
    int flags;
    unsigned int fill;
    unsigned int select;
    void *bmh;
};
```

The following describes all of the fields within the loadILBM structure and how they are used.

File

This is a pointer to a null terminated string (ASCII) of characters containing the name of the picture file to load.

bitmap1

This is a pointer to a valid bitmap structure. If you are NOT allocating one or more bitmaps with the load, then you must set this pointer to the address of an initialized bitmap structure (see `gs_get_bitmap`). If you ARE allocating one or more bitmaps with the load, then you need not initialize this field. If the load is successful, then this field, upon return, will point to the bitmap structure where the picture was loaded. Remember, if you allow the load to allocate the bitmap(s), you must free them yourself before exiting with a call to `gs_free_bitmap`.

bitmap2

If you are going to use the picture in a double buffered display, and you want the picture automatically loaded to both bitmaps, then this must contain a pointer to a second valid bitmap structure; otherwise this field must be NULL. If you are NOT allocating bitmaps with the load, then you must set this pointer to the address of a second initialized bitmap structure (see `gs_get_bitmap`). If you ARE allocating bitmaps with the load, then you need not initialize this field. If the load is successful, then this field, upon return, will point to a second bitmap structure where the picture was loaded. Remember, if you allow the load to allocate the bitmaps, you must free them yourself before exiting with a call to `gs_free_bitmap`.

color_table

If you want the load to fill a color table with the picture's color palette, then this field must point to an array of unsigned long integers (long words) that will hold the color palette upon successful completion of the load. The color table will be in standard GameSmith color format.

num_colors

If you are having the picture loader fill a color table, then you must set this field to equal the number of entries in your color table array. The load will not fill more than `num_colors` entries in the color table.

NOTE: Upon return, the loader fills this field with the actual number of colors filled in the color table. This means you should ALWAYS initialize this field before calling the loader.

dheight

This field is filled upon successful completion of the load. It contains the display height information obtained from the file. Note that this value may be different from the actual height of the picture (see the `Rows` field of the bitmap structure). This indicates the height of the Amiga display that was used while creating the picture.

dwidth

This field is filled upon successful completion of the load. It contains the display width (in pixels) obtained from the file. Note that this value may be different from the actual width of the picture (see the `BytesPerRow` field of the bitmap structure). This indicates the width of the Amiga display that was used while creating the picture.

xoff

This field is filled upon successful completion of the load. It contains the X display offset (in pixels) obtained from the file. This is usually zero, and indicates the offset from the left of the picture where the display started when the picture was being created.

yoff

This field is filled upon successful completion of the load. It contains the Y display offset (in rows) obtained from the file. This is usually zero, and indicates the offset from the top of the picture where the display started when the picture was being created.

modes

This field is filled upon successful completion of the load. It contains the display modes (mode ID) used when the picture was created.

loadx

You may optionally load a picture and offset it within the destination bitmap. This can be useful, for instance, when centering a picture file in a bitmap that is larger than the display. This is the offset value, in bytes, from the left edge of the destination bitmap. If you do not want to offset the picture within the bitmap, then set this field to zero. For instance, if you wanted to load a picture and skip the first 8 bytes on the left of the bitmap, then you would set this field to 8.

loady

If you want to offset the picture in the bitmap, this is the offset value, in rows, from the top edge of the destination bitmap. If you do not want to offset the picture within the bitmap, then set this field to zero. For instance, if you wanted to load a picture and skip the first 10 rows at the top of the bitmap, then you would set this field to 10.

flags

You may control various operations of the load by setting the appropriate bits of this field.

ILBM_COLOR Setting this flag causes the load routine to allocate one bitmap large enough to hold the picture.

ILBM_ALLOC1 Setting this flag causes the load routine to allocate one bitmap large enough to hold the picture. Upon successful completion, the bitmap 1 field will point to the bitmap structure. Remember to free the bitmap when you are finished with it.

ILBM_ALLOC2 Setting this flag causes the load routine to allocate two bitmaps of equal size, and large enough to hold the picture. Upon successful completion, the bitmap 1 and bitmap 2 fields point to the bitmap structures.

NOTE Never set both **ILBM_ALLOC1** and **ILBM_ALLOC2**. Use one or the other. Remember to free the bitmaps when you are finished with them.

fill

This is a bit mask specifying which bitplanes to fill in the destination bitmap(s). A 1 bit means that the corresponding bitplane in the destination bitmap(s) will be filled. For instance, if you want only bitplanes 0 and 2 filled, then set this value to 05 (bit 0 and 2 are set). If you want to fill all bitplanes in the bitmap, then set this value to hex ffffffff (all bits set).

This particular field comes in handy, for instance, when you have two separate images, and each needs to be loaded to a different playfield of a dual playfield display. It is a simple thing to stagger load the files, and to load the color tables into the correct offset of the display palette. For playfield 1 pic, set a fill value corresponding to odd numbered planes (for instance 01010101 binary, or 0x55 hex). For playfield 2 pic, set a fill value corresponding to even numbered planes (for instance 10101010 binary, or 0xaa hex).

select

This is a bit mask specifying which bitplanes to load from the picture file. This allows selective bitplane loading from an IFF ILBM image. A 1 bit means that the corresponding bitplane from the picture will be loaded. For instance, to load only bitplane 1 from a picture file, set this value to 02 (only bit 1 is set). If you want to load all bitplanes in a picture, then set this value to hex ffffffff (all bits set). You can shuffle and depth arrange bitplanes by using the fill and select fields.

bmh

This is a pointer to a BitMapHeader structure (defined by Commodore), which encompasses all information located in the BMHD chunk of an IFF ILBM file. If this pointer is NULL, then it is simply ignored by the loader. Otherwise, the loader assumes that it points to a valid BitMapHeader structure and fills all fields accordingly. This is provided in case your program needs additional information about the ILBM file.

The BitMapHeader Structure

The BitMapHeader structure defined by Commodore, and optionally filled by the ILBM loader function is as follows:

```
struct BitMapHeader {
    UWORD w,h;           /* raster width height in pixels */
    WORD x,y;            /* same as xoff & yoff of load ILBM */
    UBYTE nPlanes;       /* number of bitplanes */
    UBYTE masking;       /* mask value */
    UBYTE compression;   /* compression algorythm */
    UBYTE reserved;
    UWORD TransparentColor;
    UBYTE xAspect,
    UBYTE yAspect;       /* pixel aspect ratio width to height) */
    WORD pageWidth,
    WORD pageHeight;     /* same as dwidth & dheight */
};
```

For detailed information on the fields in this structure, refer to Commodore documentation on the ILBM format. You will probably find the w and h fields most useful, as they describe the exact pixel width and height of a picture. The bitmap structures BytesPerRow will often be a rounded up number so that it is an even multiple of 16 or 32.

The Bitmap Structure

Literally ALL GameSmith graphics revolve around the bitmap structure defined by Commodore Amiga. The GameSmith graphics routines do all of their own drawing to the bitmap, allowing them to remain compatible with all versions of the operating system, as well as allowing their use in conjunction with standard system graphics like the Intuition windowing environment. In other words, you may plot points, blit images, and animate complex objects just as easily within an Intuition Screen or Window as you can in your own custom display. In addition, direct use of the bitmap gives the GameSmith system it's high speed graphics.

The bitmap structure defined by Commodore Amiga is as follows:

```
typedef UBYTE *PLANEPTR;
```

```

struct BitMap
{
    UWORD BytesPerRow;
    UWORD Rows;
    UBYTE Flags;
    UBYTE Depth;
    UWORD pad;
    PLANEPTR Planes[8];
};

```

This simple structure contains all the information necessary for drawing graphics, and is also the basis behind all of the Amiga's internal graphics routines. All Intuition Screens and Windows have an associated bitmap, as do the graphics primitives View and ViewPort structures. The GameSmith graphics routines do not, as yet, support interleaved bitmaps, but will in the future.

Currently, the bitmap structure is limited to 8 bitplanes, restricting the direct use of 24 bit graphics boards. This structure will probably be enhanced in the future with the addition of 16 plane pointers and an extra flag. When this happens, the GameSmith graphics system will directly support 24 bit output.

The GameSmith Color Table

Every GameSmith function that accepts a color table requires it to be in the GameSmith format. The GameSmith anim file loader will generate an old 4 bit color table upon specific request, but in general, all GameSmith functions that generate a color table produce it in the 8 bit GameSmith color table format.

Each entry in the GameSmith color table is an unsigned long integer (long word). Usually it is in 8 bit format, meaning each value for red, green, and blue has 8 bits of color information respectively. The high byte of this format is currently unused, and should be set to 0.

'C' notation: 0x00rrggbb
 Assembler notation: \$00rrggbb

When requesting a 4 bit color table from the anim file loader, each entry still contains an unsigned long integer, but with only 4 bits of color information per red, green, and blue value. This format is directly usable with the old LoadRGB4 and SetRGB4 system function calls.

'C' notation: 0x000000rgb
 Assembler notation: \$000000rgb

Blitter Operation

The GameSmith Development System provides many low level functions for controlling the Amiga's blitter chip directly. These are a set of highly efficient routines for blitting graphics to a bitmap. In fact, the GameSmith animation system calls these functions directly. It should be noted that the blitter routines are "depth safe", in that they will not blit to nonexistent bitplanes of a bitmap. This means that blitter or animation, objects may be displayed in ANY bitmap safely, regardless of depth.

The blitter structure is the lowest level graphics structure used in the GameSmith graphics hierarchy. Every blitter function requires that graphics data be referenced through a blitter structure.

NOTE: Usually, you will set up your graphics objects using CITAS. CITAS automatically generates the blitter structure, so you almost never need to be concerned with its contents. This is especially true when working with the animation system and animated objects. Your program need only reference the animated object by a pointer to its controlling structure.

```

struct blit_struct
{
    unsigned short *data;
    unsigned short *mask;
    unsigned short *save;
    int depth;
    int planes;
    int width;
    int height;
    int image_size;
    int x_off;
    int y_off;
    int flags;
    struct coll_bg_struct *coll_bg;
    struct collision_struct *collision;
    struct blit_struct *prev;
    struct blit_struct *next;
    int reserved[2];
};

```

The following describes all of the fields within the blitter structure and how they are used.

data

This is a pointer to the actual graphics data, which may reside in either Fast RAM or Chip RAM depending on the status of the BLIT_CPUBLIT flag (see below). All graphics data (except the mask) must be contiguous in memory starting with the first plane (from 0 to n), although the first plane of data need not be displayed in bitplane 0 of the bitmap (see the planes field).

mask

This is a pointer to the image mask required by some blitter functions. The mask defines the area of the image that will be copied to the target bitmap. This is akin to a cookie cutter, or a paper doll pattern. The image mask is simply a bitwise OR of all bitplanes in the image.

save

This is a pointer to a save area of memory, the same size as the image data, for saving background graphics "behind" a blitter image. Again, save areas are allocated from either Fast RAM or Chip RAM depending on the status of the BLIT_CPUBLIT flag (see below). The save operation takes place before the image is actually blitted to the bitmap.

Restoring the background has the effect of erasing the image from the screen. A repeated combination of saving and restoring the background allows an image to move "on top of" a background picture. Note that not all blitter functions can save background data. Also, there are specific GameSmith functions for obtaining and freeing blitter save areas.

depth

This is the number of bitplanes that make up the image.

planes

This is a bit mask of the planes that should be affected in the target bitmap. A 1 bit in the planes mask means that the corresponding bitplane in the target bitmap should be written to. In other words, if bit 0 is set (value of 1) in the planes mask, then bitplane 0 of the target bitmap should be written to with a plane of image data. For instance, if an image is made up of 3 bitplanes and has a planes mask of 0x16 (00010110 binary), then planes 1, 2, and 4 will be affected in the target bitmap. Typically, unless the merge bit is set, the unaffected planes of the bitmap under the image mask are cleared to zero. This allows the image to appear with the correct colors. The correct use of this field allows the blitter routines to save time and space. NULL image planes do not have to be kept in memory (possibly chewing up precious Chip RAM), and it allows the blitter to use a higher speed clear operation instead of copying a NULL plane.

CAUTION! Never set this field to all 1's if you don't have 32 planes of image data. When blitting to a bitmap deeper than the image, garbage will get mixed in with the image. There should be one bit set in the planes mask for every valid plane of image data, and no more.

width

This is the image width in 16 bit words (unsigned shorts).

height

This is the height of the image in scan lines.

image_size

This is the size, in bytes, of one bitplane within the image. This is computed by taking the width (in bytes) times the height.

x_off

This is a positive or negative offset from the destination X coordinate at which the left edge of the image will start when copied to a target bitmap. In other words, if the programmer is placing an image at X coordinate 100, and the *x_off* field is 2, then instead of the left edge of the image being at 100, it is copied beginning at 102.

y_off

This is a positive or negative offset from the destination Y coordinate at which the top edge of the image will start when copied to a target bitmap. In other words, if the programmer is placing an image at Y coordinate 100, and the *y_off* field is 2, then instead of the top edge of the image being at 100, it is copied beginning at 102.

flags

The blitter flags are as follows:

BLIT_MERGE	Merge image data with background data, do not clear unaffected planes. This causes translucency effects.
BLIT_1SHOT_FORWARD	This is a CITAS generated flag used when loading an animated disk object. Do not set or clear this bit.

BLIT_1SHOT_BACKWARD	This is a CITAS generated flag used when loading an animated disk object. Do not set or clear this bit.
BLIT_DATACOPY	Image data is a copy area. DON'T TOUCH!
BLIT_CPUBLIT	Use the CPU to blit the image data instead of the Amiga's blitter chip.

coll_bg

This is a pointer to a background collision detection structure. The animation system uses this structure to determine if an image is colliding with background graphics data. If this field is NULL, then the image cannot collide with background graphics. See Chapter 5 The Animation System for more information.

collision

This is a pointer to an object-to-object collision detection structure. The animation system uses this structure to determine if one image is colliding with another. If this field is NULL, then it cannot collide with other images. See the chapter on animation for more information.

prev

This is a pointer to the previous image in an animation sequence. This is used by the animation system. If this field is NULL, then this image is the first in the animation sequence.

next

This is a pointer to the next image in an animation sequence. This is used by the animation system. If this field is NULL, then this image is the last in the animation sequence.

CPU Blittable Objects

If you read through the previous section, you no doubt noticed mention of the CPUBLIT flag. Every single blitter function except the blitter memory copy has a CPU equivalent. Operation is transparent, depending on the status of the BLIT_CPUBLIT flag in the blitter structure (see above). Anim objects that use the CPU for display must also have the ANIM_CPUBLIT flag set (see the chapter on animation). CITAS allows you to use this capability with the mere click of a button. What good is this option? Here are a couple good reasons why you might want to take advantage of this capability:

- CPU blittable object data and save areas may be located in Fast RAM, thus conserving valuable Chip RAM. Even on slow systems, this has numerous applications - such as adventure games, or any application that doesn't require fast and furious graphics.
- On systems where the CPU can perform blit operations faster than the Amiga blitter chip, it provides an even faster method of image display. Note, however, that the AGA blitter still performs blit operations significantly faster than even a 25 MHz 68040!

User Copper Lists

The following is a discussion of an advanced graphics topic. Those that are not very familiar with Amiga graphics and copper concepts may be better served to learn the rest of the system first with the knowledge that you can come back and study this later.

Setting up a table of copper instructions and adding them to any display is accomplished through the GameSmith copper structure. User copper lists for a GameSmith display are built and maintained completely independent of the operating system. Within the GameSmith display, the user copper list is transformed into a hardware list which is directly executable by the copper chip. These lists are separate from the main GameSmith display list, which contains all of the display control stuff the user will typically want to avoid messing with. The main display list actually makes a function call to the user list on a per viewport basis!

It can also be used to change an existing user copper list for dynamic effects. The copper instructions currently available in the Amiga graphics system are well for creating fantastic displays such as a rainbow colored background. All of this is accomplished through a table of copper commands pointed to by the copper structure.

The Copper Structure

```
struct copper_struct {
    unsigned short *list;      /* pointer to copper list */
    void *copctrl;             /* pointer to UCopList control struct */
    void *display;             /* pointer to appropriate display struct */
};
```

The following describes all of the fields within the copper structure and how they are used.

list

This is a pointer to the list of copper commands.

copctrl

This is a pointer to a system UCOPLIST structure. Normally, you will fill this field with NULL and never worry about it. Only when changing the copper list of an existing display do you need to fill this field.

display

If you are passing a copper list to the `gs_create_display` function, simply fill this field with NULL and ignore it. Otherwise, if you are calling `gs_user_copper` yourself you will need to fill this field with the appropriate controlling structure for the display you're modifying. This will be either a pointer to an Intuition Screen struct, or a pointer to a ViewPort struct, depending on the type of display.

Copper Commands

UC_MOVE	Move a value into a hardware register. The syntax for this is UC_MOVE,reg,value. The reg is the hardware offset for the register (such as 0x96 is the hardware offset for the DMA control register. Refer to the Amiga Hardware Reference Manual for complete details). Value is the 16 bit value you want placed in the hardware register
UC_WAIT	Wait for the video beam to reach a specified Y,X coordinate before executing the next command in the list. Syntax is UC_WAIT,y,x. Note that all X and Y values assume a low resolution display. Again, see the hardware manual for details.
UC_SETCOLOR	Set a color register. Syntax is UC_SETCOLOR,color_number,color_value. Color values are in old 4 bit format (4 bits each for red, green, and blue). This can only affect color registers 0 through 31.
UC_END	This marks the end of the copper list This command MUST be the last command in the table.

Modifying an Intuition Display

To modify an existing Intuition screen, follow these steps:

- Issue a `Forbid()` call.
- Set the list pointer of the copper structure to point to your command table.
- Set the `copctrl` pointer to that of the screen's `UCopList` structure. Copy the screen->`Viewport.UCoplus` field.
- Set the display pointer of the copper structure to point to the Screen structure.
- Call `gs_user_copper`
- Issue a `Permit()` call.
- Call `RethinkDisplay()` to show the change.

Because `gs_user_copper` was written in 'C', assembler programmers will need to push the function parameters on the stack before making this call. Also, see the include file `GameSmith.i` for exact structure field names.

Modifying a GameSmith Display

Once a hardware list has been built from the mnemonic user copper list, your program can directly modify it's contents for stunning and immediate special effects!! You can also swap entire lists quickly and easily.

There are two ways to install a user copper list on a GameSmith display:

- Place the address of your mnemonic list in the `ucop` field of the `gs_viewport` structure before creating the display, and let the GameSmith display system handle it from there. Note that this field no longer points to a `copper_struct`, but to the array of copper commands itself.
- Call the function `gs_hard_copper()` with the address of a `hard_copper` structure, and install the new list with a call to `gs_replace_ucop()`. The function `gs_replace_ucop()` is very fast, and enables the user to swap between different copper lists at a rapid pace. This method of copper list modification takes a little more explanation.

If you intend to swap many lists, here's how it's done:

- Call `gs_hard_copper()` to build all of your lists (remember to replace the list pointer before each call).
- After each call, save the following fields returned in the `hard_copper` structure: both `LOF_cop` pointers, both `SHF_cop` pointers, and the `coplen` value.
- Replace the above mentioned fields in the `hard_copper` structure with the save values for the list you want to show on the display, and call `gs_replace_ucop()`.
IMPORTANT! You must save the fields again after the 1st call to `gs_replace_ucop()`. These values are those initially allocated when the display was built, and must be freed as well before your program exits.
- Before exiting your program free all lists EXCEPT the one the display is currently using by calling `gs_free_hard_copper()`.

CAUTION: Never call `gs_free_hard_copper()` for the lists which the display is using (the last ones installed with `gs_replace_ucop()`). The display system itself will free these when you call `gs_remove_display()`.

The Hard Copper Structure

```
struct hard_copper {
    struct display_struct *d; /* ptr to display structure */
    ushort *list;             /* ptr to list of copper instructions (mnemonic list) */
    ulong flags;              /* hard copper flags (allocation, etc.)* */
/* the following are system only fields. Do not touch! */
    ushort *LOF_cop[2]; /* ptr to hardware copper list pgs 1 & 2 */
    ushort *SHF_cop[2]; /* ptr to hardware copper list pgs 1 & 2 */
    ulong coplen;         /* length (in bytes) of hardware copper list */
};
```

Fields which the user is concerned with, and should initialize:

d

Pointer to a GameSmith display which has already been created.

list

Pointer to the user copper instruction list, which is an array of unsigned short integers (16 bit words).

flags

UCF_DOUBLE Build two display lists for a double buffered display. If you need your list installed in a double buffered display, then you should set this flag bit. The GameSmith copper system will then generate two copies of your list, and `gs_replace_ucop()` will automatically install the proper lists in the proper places. Use of this bit also causes the 2nd bitmap pointer of a pvp or spvp structure to be used with page 2 of the display.

New User Copper Instructions:

UC_SETCOLORAGA	Set a 24 bit AGA color register Format: UC_SETCOLORAGA,colornum (0-255), red (8 bit), green (8 bit), blue (8 bit)
UC_PV	Load a parallax viewport format: (see above)
UC_SPVP	Load a sliced parallax viewport format: (see above)

Breakdown of instructions you may modify in the hardware list (see also the Amiga Hardware Reference Manual for instruction details):

UC_WAIT	word 1, high byte = Y, or vertical beam position (scan line) word 1, low byte = X, or horizontal beam position. Bit 0 of this byte must always be set to 1. word 2 = 0xfffe Don't change word 2.
UC_MOVE	word 1 = Hardware register to affect. Bit 0 of this word must always be 0. word 2 = 16 bit data value to be transferred to destination register.
UC_SETCOLOR	word 1= color register to affect (0x180+(colornum*2)) word 2 = color value in the format 0x0rgb (4 bits of red, green, blue respectively).
UC_SETCOLORAGA	1st 6 bytes = color bank select. Don't change this. word 4 = high 4 bits of 0x0rgb respectively. 6 bytes = color bank select. Don't change this. word 8 = low 4 bits of 0x0rgb respectively.

Copper Instructions Lengths

UC_END	12
UC_WAIT	4
UC_MOVE	4
UC_SETCOLOR	4
UC_SETCOLORAGA	16
UC_PVP	20+(pvp->depth*8) If dual playfield reload (DPF1 or DPF2) then subtract 4 bytes from this value.
UC_SPVP	spvp->height*(20+(spvp->depth*8)) If dual playfield reload (DPF1 or DPF2), then subtract (4*spvp->height) from this value. Also, if reloading color registers every line, then add (height*spvp->nregs*4) for 12 bit reloads, and (height*spvp->nregs*16) for AGA 24 bit reloads, where height is spvp->height or 1 if PVP_LINE1 is set in the spvp flags.

Graphics Functions

<u>Function</u>	<u>Description</u>
gs_loadILBM	Load an IFF ILBM picture file from disk
gs_get_ILBM_bm	Examine an IFF ILBM picture file and fill a bitmap structure accordingly
gs_chiprev	Inquire chipset version
gs_plot	Plot a point in a bitmap using the specified color
gs_plot_reverse	Exclusive OR a point in a bitmap
gs_plot_test	Get the color value of a point in a bitmap
gs_get_blitter	Allocate the blitter in a system friendly way
gs_free_blitter	Allow other tasks to use the blitter
gs_blit_image	Copy a graphic image to a bitmap using a mask
gs_blit_image_fine	Copy a graphic image to a bitmap using a mask and fine control variables
gs_blit_image2	Copy a graphic image to a bitmap using a mask after first calling gs_blit_save_bg
gs_blit_copy	Copy a graphic image to a bitmap without a mask
gs_blit_copy_fine	Copy a graphic image to a bitmap with fine control variables (no mask)
gs_blit_copy2	Copy a graphic image to a bitmap without a mask after first calling gs_blit_save_bg
gs_blit_save_bg	Save background data from a bitmap where an image will be placed
gs_blit_restore	Restore background data
gs_blit_clear	Remove an image from a bitmap by writing zeros over the image
gs_blit_get_save_area	Allocate a save area for background data
gs_blit_free_save_area	Release save area memory
gs_blit_memcpy	Copy a block of memory using the blitter
gs_blit_fill	Fill a Chip memory area with a value
gs_get_bitmap	Allocate a bitmap
gs_free_bitmap	Release a bitmap
gs_user_copper	Attach or change a user copper list in a display
gs_hard_copper	Build a hardware list out of a user copper list
gs_replace_ucop	Replace the current user copper list
gs_free_hard_copper	Free the list pointed by the LOF_cop and SHF_cop

4 The Display System

The GameSmith display system provides easy access to the Amiga's low level graphics routines. It handles display setup and control, which can involve many complex issues. Note, however, that this display system takes over the entire Amiga display. This is not generally considered a system friendly thing to do, as the Intuition screens will no longer be visible (until you remove the GameSmith display). This is generally OK for games, however, and is often necessary for speedy display updates. The GameSmith display system also provides a way to do this in a system friendly way that still permits fast game operation in a multi-tasking environment.

The GameSmith display system makes use of the Amiga's system View and ViewPort display handling. Once the operating system has constructed the display copper lists, the GameSmith display system takes over from there. This sort of hybrid approach provides the greatest compatibility with all machines. The GameSmith display system provides super fine hardware scrolling, multiple independently controllable viewports, separate dual playfield scrolling, and advanced AGA modes like 8 bitplane dual playfield control.

For some games, such as the fantasy adventure type (dungeon type games), you will probably want to use a simple Intuition screen for your display.

For complete information on the Amiga's View and ViewPort display handling, refer to the Amiga ROM Kernel Reference Manual: LIBRARIES.

Double Buffering

A simple technique called double buffering allows you to create completely smooth animated graphics. It works by displaying one display "page", or buffer, while drawing into a second page. When drawing is complete the second page is displayed while the first one gets updated. In this way, the user does not see the drawing take place, and the eye sees only smooth animation or movement on the screen. This technique is also referred to as page flipping.

The drawback to this technique is that it requires two complete displays in memory instead of just one. However, it is the only way to get rock solid smooth animations.

The GameSmith display system provides support for a double buffered display, and the GameSmith animation system can handle two simultaneous bitmaps so that your program need never worry about it. Most of the included examples make use of a double buffered display.

You can also double buffer an Intuition display. Page flipping is slower, but the technique is a simple one. See the example code for a demonstration of this technique. In fact, CITAS itself makes use of this technique during animation preview.

The Display Structures

The GameSmith display system requires properly initialized display structures. The display structures contain all necessary information on how to build the display. If you call the function `gs_display` instead of `gs_create_display`, then the display structures are allocated and built for you, and returned to your program if the call is successful. The function `gs_display` calls `gs_create_display`.

The Rectangle Structure

```
struct gs_rectangle
{
    short MinX,MinY;
    short MaxX,MaxY;
};
```

This structure is identical to Commodore's Rectangle structure. It is used to ensure that people with older systems will still be able to compile the header file correctly. The fields within this structure define the left, top, right, and bottom of a rectangle respectively.

The Viewport Structure

```
struct gs_viewport
{
    struct gs_viewport *next;
    unsigned long *color_table;
    int num_colors;
    short int *ucop;
    int height;
    int width;
    int depth;
    int bmheight;
    int bmwidth;
    int top;
    int left;
    int xoff;
    int yoff;
    unsigned long flags;
    void *vpel;
    void *vpe2;
    struct BitMap *bitmap1;
    struct BitMap *bitmap2;
    void *extension;
    struct gs_rectangle dclip;
    struct ViewPort *viewport1;
    struct ViewPort *viewport2;
    struct RasInfo *rasinfo1;
    struct RasInfo *rasinfo2;
    /* other GDS only fields */
};
```

The `gs_viewport` structure contains information about how to set up and display a controllable "window" on the screen. The viewport may be the entire height and width of the screen, or may only be a portion of it. You can split the screen horizontally, and have more than one viewport in the display.

The following describes each field within the `gs_viewport` structure and how it is used by the display software:

next

This is a pointer to the next `gs_viewport` in the display. If this is the last one in the display, then this pointer must be NULL.

color_table

Table of colors that define the color palette for this viewport. Color table entries must be in standard GameSmith format.

num_colors

This field contains the actual number of colors in the color table. If your display contains more colors than supplied in the color table, then color values for the additional color registers is undefined.

ucop

This is a pointer to an array of unsigned short integers (16 bit words) which contain a mnemonic copper instruction list. If you have no user copper list for your display, simply fill this field with NULL.

height

This is the height of the viewport in scan line.

width

This is the width of the viewport in pixels.

depth

This is the number of bitplanes to use for the viewport.

bmheight

This is the height of the bitmap in rows, or scan lines. This field is only used when the display system is to allocate the necessary bitmap(s) for the display. A bitmap can be larger than the display size (height and/or width). If the bitmap is larger than the display, the viewport can be scrolled (see the section on scrolling below).

bmwidth

This is the width of the bitmap in pixels. This field is only used when the display system is to allocate the necessary bitmap(s) for the display. A bitmap can be larger than the display size (height and/or width). If the bitmap is larger than the display, the viewport can be scrolled (see the section on scrolling below).

top

This is the offset, in rows, from the top of the display for the start of the viewport. This value must never be negative. Viewports may be positioned anywhere within the actual display. The first viewport in a display should have a top offset of zero, meaning that it begins at the top of the display area. The second (if any) would have a top offset at least as great as the previous viewport's height + 1 (see the section on multiple viewports below).

left

This is the offset, in pixels, from the left of the display for the start of the viewport. This value must never be negative. Viewports may be positioned anywhere within the actual display. Usually, this value is zero, meaning that the viewport starts at the first displayable position at the left of the display area.

xoff

This is the X offset, in pixels, where the bitmap display begins. This value should never be negative. This field is only relevant to bitmaps which are wider than the actual viewport display area. See the section on scrolling below for information on setting up the xoff value.

yoff

This is the Y offset, in rows, where the bitmap display begins. This value should never be negative. This field is only relevant to bitmaps which are taller than the actual viewport display area. See the section on scrolling below for information on setting up the yoff value.

flags

The GameSmith viewport flags are as follows:

- | | |
|--------------|--|
| GSVP_ALLOCBM | Let the display setup routines allocate the necessary bitmap(s) for the viewport. Note that the fields bmheight and bmwidth MUST be filled with the dimensions you require. Also, the display setup calls <code>gs_get_bitmap</code> to allocate the bitmap(s) and places the pointer or pointers in the bitmap 1 and bitmap 2 fields of the <code>gs_viewport</code> structure. Note that if you use this flag to allocate the bitmap(s), you should NOT free the bitmaps yourself; <code>gs_remove_display</code> will take care of the necessary cleanup. |
| GSVP_DCLIP | Use the specified display clip in the dclip rectangle. This pertains ONLY to systems running Workbench 2.0 or above, and affects the DxOffset and DyOffset of the View itself. If a custom display clip is not specified, then the display system will use the standard display area for the given mode, set up by the user in system Preferences. Dimension Info for the display clip may be obtained by calling <code>GetDisplayInfoData</code> . |
| GSVP_DPF | Display the viewport in dual playfield mode. |
| GSVP_NOCOLOR | This tells the GameSmith display not to load a color palette for the viewport in the main copper list. |
| GSVP_NOWAIT | This bit is relevant to the first viewport only, and tells the GameSmith display system not to build a copper wait instruction which waits for the top of the display window. This is useful when you need your own user copper list to execute before the actual start of display data. |

vpel, vpe2

These fields are maintained by the display system. They point to one or two ViewPortExtra structures when operating under Workbench 2.0 or above. You need not be concerned with these fields.

bitmap1, bitmap2

If you are allowing the display system to allocate the bitmap (s) for you (GSVP_ALLOCBM flag set), then you need not be concerned with these fields before calling a display setup function. Upon successful return of the display setup, these fields point to one or two valid bitmap structures, depending on whether the display is double buffered or not (see the GSV_DOUBLE flag of the display structure).

If you have already obtained the bitmap(s) through other means (`gs_get_bitmap`), then these fields must point to the bitmap structures you want the display to use. If the display is only single buffered, then you should clear the second bitmap pointer to NULL.

extension

This field is reserved for future use. For compatibility, you should fill this field with a NULL.

dclip

If your software is running under Workbench 2.0 or above, then you may opt to specify your own display clip. Display clip dimensions for the various standard and overscan areas may be obtained by calling `GetDisplayInfoData` to fill a Dimension Info structure. Note that all except the video overscan clip are user defined via Preferences, so your display should appear in the correct position on the user's screen. If a custom display clip is not specified, and your software is running under Workbench 2.0 or above, then the display system will use the standard "nominal" display area for the given mode.

viewport1, viewport2

These fields point to actual system ViewPort structures. After the display is set up, you may reference these structures. Note that the `viewport2` field will only be valid if you are using a double buffered display.

rasinfo1, rasinfo2

These fields contain pointers to actual system RasInfo structures. After the display is set up, you may reference these structures. Note that the `rasinfo2` field will only be valid if you are using a double buffered display.

The Display Structure

The following describes the display structure. This is the structure to which all viewports in a display are chained, and can be considered the overall controlling structure for a display.

```
struct display_struct
{
    struct View *oldview;
    void *vel;
    void *ve2;
    int DxOffset;
    int DyOffset;
    unsigned char sprpri_pf1;
    unsigned char sprpri_pf2;
    int modes;
    unsigned long flags;
    struct gs_viewport *vp;
    void *extension;
    struct View *view1;
```

```

struct View *view2;
/* other GDS only fields */
};

```

oldview

This field will contain a pointer to the previous View structure that was displayed prior to displaying a custom GameSmith View. The old View will generally be the Intuition Screen system.

ve1, ve2

These fields will contain pointers to system ViewExtra structures upon successful display setup (Workbench 2.0 or above only). You need not be concerned with these fields.

DxOffset, DyOffset

This field provides a Workbench 1.3 compatible way of specifying the DxOffset and DyOffset for a View.

sprpri_pf1, sprpri_pf2

These must contain the sprite priorities relative to playfield 1 and 2 respectively. See the Amiga Hardware Reference Manual for details on sprite priorities with respect to playfields. As per the manual, valid values range from 0 to 4. The following illustrates the meaning of these priority values (valid for both playfields on an independent basis):

Frontmost	0	Playfield display sprites 0 & 1
	1	sprites 2 & 3
	2	sprites 4 & 5
	3	sprites 6 & 7
Bottommost	4	
(behind others)		

To have all sprites appear in front of the given playfield, use a value of 4.

modes

This field must be filled with the modes you wish for your display (hires, lace, superhires, halfbright, HAM, etc.). For Workbench 1.3, there are only a few modes, but for 2.0, and especially 3.0, the possible modes are numerous. Under Workbench 2.0 and above, this field is interpreted as the mode ID for the display.

flags

The display flags you may set or read are as follows:

GSV_DOUBLE Set this flag to set up a double buffered display.

GSV_FLIP This bit is set by calling `gs_flip_display`, and is cleared when the new page is actually shown. Your program may wait for this bit to clear after calling `gs_flip_display`, and before doing any additional drawing in a double buffered display.

GSV_SCROLL1	This bit is set by the system when making a synchronized scroll. It indicates that page 1 copper lists should be reloaded with new scroll values during the next vertical blank period. Your program may wait for this bit to be cleared before making another call to the scroll routines.
GSV_SCROLL2	This bit is set by the system when making a synchronized scroll. It indicates that page 2 copper lists should be reloaded with new scroll values during the next vertical blank period. This bit is only set for a double buffered display.
GSV_SCROLLABLE	Set this flag to allow the viewports in the display to be scrolled. See the section on scrolling below for an explanation of why this bit is necessary.
GSV_DDFADJUST	This is set by the display system if it was able to adjust the data fetch to allow smooth scrolling instead of the display window start (see section on scrolling below). This bit will only be set on AGA systems running "normal" (lores & hires) unpromoted modes.
GSV_HARDX	Setting this bit tells the GameSmith display system to use the supplied DxOffset even under operating system 2.0 and above. The DxOffset field contains a value which specifies where the left edge of your display starts. Normally this is determined by the users preferences setting for the given display mode. Due to problems with the Amiga display hardware when the screen starts too far to the left, any application which uses horizontal scrolling should set this bit so that the display will start at the same position on all computers. However, this value needs to be different depending on the chipset under which the program is running (see the function <code>gs_chiprev()</code> below). The following values center the display nicely under the various conditions: AGA - 0x85 (\$85 assembler) ECS - 0x77 (\$77 assembler) OCS - 0
GSV_HARDY	Use the supplied DyOffset even under 2.x+ to specify the top start of display.
GSV_PF2PRI	Normally in a dual playfield display, playfield 1 shows in front of playfield 2. Setting this bit enables playfield 2 to appear in front of playfield 1.
GSV_BRDRBLNK	This bit, when set, will blank the borders on AGA and ECS machines instead of showing color 0 (border area will be black).

The manual states that when making a scrollable display under ECS/OCS, some of the viewable width that your program asks for is hidden on the left to allow for horizontal scrolling. This is no longer true under a lores display. What you ask for is what you get, so the visible area is the same on all machines.

vp

This field must point to the first `gs_viewport` structure for the display.

extension

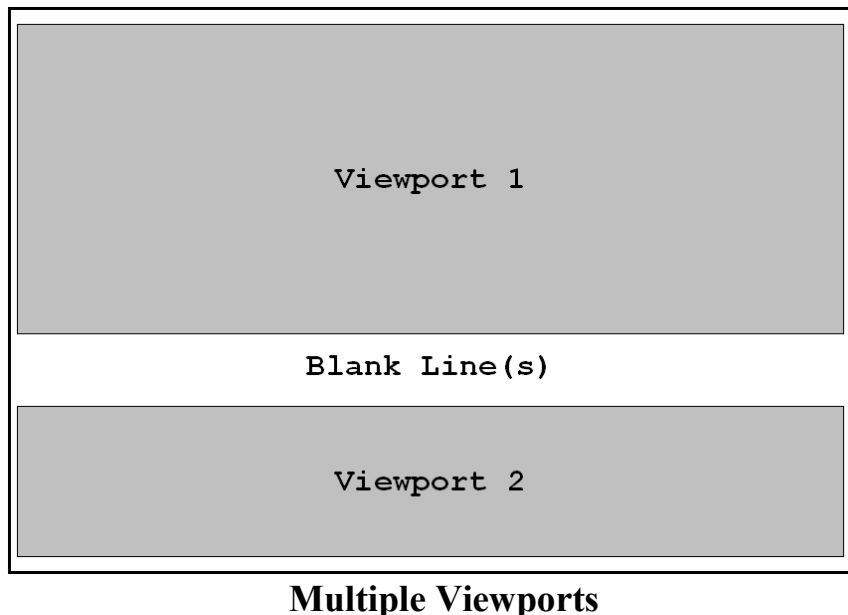
This field is reserved for future use. For compatibility, you should set this field to NULL.

view1, view2

These fields point to actual system View structures. After the display is set up, you may reference these fields as valid system structures.

Multiple Viewports in a Single Display

The Amiga system allows you to have multiple, vertically stacked, viewports per display, each with its own bitmap, color palette, and copper list. The system requires there be at least one blank (unused) horizontal scan line between each viewport on the screen. Depending on display resolution and the number of bitplanes used in each viewport, the system may require additional blank lines as "breathing room". The system uses the time between viewport displays to get set up to display the next viewport on the screen.



If you experience strange side effects, particularly in graphics display at the top of a viewport, then try adding more space between the viewports. This should clear up the problem. Note also that each viewport in the display may optionally be shown in dual playfield mode by setting the GSVP_DPF flag bit in the `gs_viewport` structure. This is the only mode change allowed within the display. All other mode ID's are constant throughout the entire display.

Scrolling

The GameSmith display system allows a viewport to be scrolled in 4 different directions: up, down, left, and right. Because the scroll functions accept both a delta X and a delta Y value (change in offset), you can actually scroll the display at any angle. For instance, scroll values of 1, -1 would scroll the viewport up and to the right at a 45 degree angle.

When the edge of the actual bitmap is reached, scrolling will automatically stop. This is a nice fail-safe feature. See the description of the `gs_scroll_vp` functions for a list of possible return values.

Note that in order for a viewport to be scrollable, its associated bitmap must be larger than the size of the viewport. That is, if you have a viewport with a bitmap that is taller than the viewport height, you may scroll that viewport up and down. If you have a viewport with a bitmap that is wider than the viewport width, you may scroll that viewport left and right.

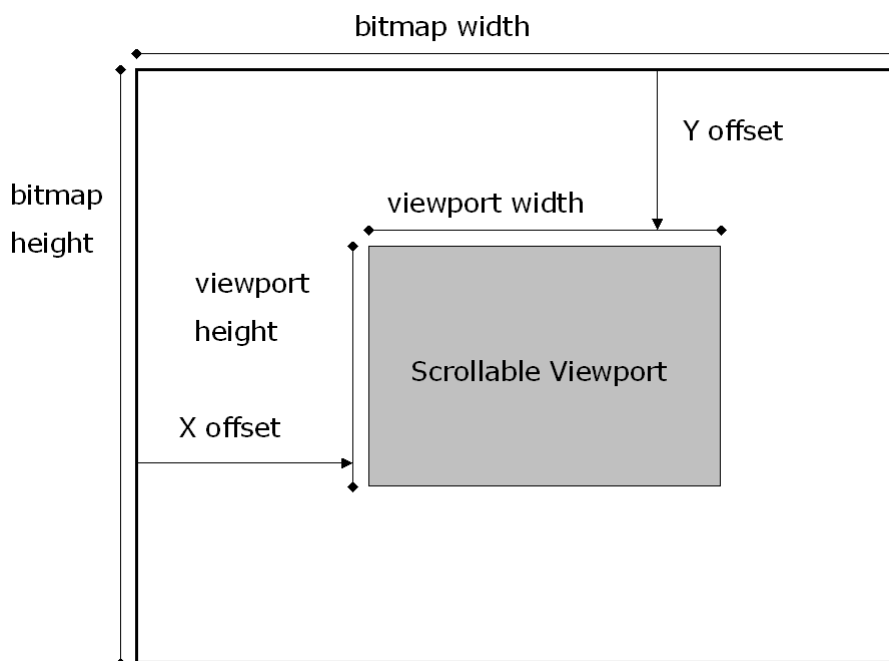
You will need to set the GSV_SCROLLABLE flag bit in the display structure in order to prevent the left border from "jumping" while scrolling a viewport horizontally. By setting this bit, the display system will adjust the left window border inward by a magic number to prevent this "jumping" effect. This is a hardware limitation. The reason for this is so the display hardware has enough time to prefetch data to the left of the display and "delay" it by a certain amount, depending on the horizontal scroll value. This is how the Amiga accomplishes its ultra-smooth sideways scrolling. At any rate, scrollable displays generally have to give up some bitmap pixels on the left of the display. This area cannot be viewed.

When running "normal" unpromoted (NTSC_MONITOR_ID or PAL_MONITOR_ID) modes (lores or hires) on an AGA system, it is usually possible to adjust the display so that you don't lose any usable pixels on the left side. When this is the case, the GSV_DDFADJUST flag bit will be set in the display structure upon successful display setup.

The number of bits lost on the left of a scrollable screen depends on the screen mode. These are shown below:

lores non-AGA machines:	16 bits
lores mode promotion (AGA):	44 bits
hires non-AGA machines:	32 bits
hires mode promotion (AGA):	10 bits
super hires/super72 (AGA):	72 bits

Scrollable Display Diagram:



AGA Mode Considerations

The AGA display hardware is a complex sort of beast, with many quirks. The unfortunate result of this is that special consideration must be given when creating an AGA compatible display.

Mode Promotion

Mode promotion is a wonderful thing. It makes use of the monitors DBLNTSC or DBLPAL to "promote", or scan double a "normal" lores or hires display, thus producing a rock solid screen. This not only prevents flicker, it "fills in" the gaps between noninterlaced display lines. This "fill in" is sometimes undesirable on a lores display, as it can make the pixels look blocky. In addition, dual playfield mode is not possible on a promoted display.

To prevent mode promotion on a lores or hires display, you'll need to use either NTSC_MONITOR_ID or PAL_MONITOR_ID in the modes field of the display structure. You can OR, in the HIRES and LACE keys as desired (see the Commodore include file MODEID.H in the graphics subdirectory for a list of available monitors and modes. These monitors are only available starting with WB 2.0, but since AGA machines require at least 3.0, you can use the graphics function ModeNotAvailable to determine which type of system your program is running on: NTSC or PAL. Of course your program should check the library version for at least 2.0 before making this function call and using one or the other monitor. Look in GfxBase>LibNode.lib_Version for the current release level (after opening the graphics library). Check the included example programs for a demonstration of this technique.

Scrolling

For scrollable displays, the display clip MinX value MUST be 0 or the left border will jump when scrolled. This may be a hardware limitation of some kind, and is being investigated. For now, a value of 0 will prevent left border jump. If you don't set up a display clip (don't set the GSVP_DCLIP flag bit), then the nominal mode values will be used. These also use a MinX of 0.

Dual Playfield

The GameSmith display system allows the use of the new enhanced 7 or 8 bitplane dual playfield modes. However, this increased depth capability means that the color palette for the 2nd playfield must begin at a different place in memory. The table below shows how palette selection is made for playfield 2 depending on total bitmap depth. Note that the playfield 1 palette offset always starts with color 0.

<u>Planes</u>	<u>Starting Color Offset</u>
1 - 6	8
7 - 8	16

Here is a full list of color register usage in the new dual playfield depths:

PLAYFIELD 1

Plane Bits Color Register

0000	color 0 (transparent, playfield 2 shows through)
0001	color 1
0010	color 2
0011	color 3
0100	color 4
0101	color 5
0110	color 6
0111	color 7

AGA Depth From Here

1000	color 8
1001	color 9
1010	color 10
1011	color 11
1100	color 12
1101	color 13
1110	color 14
1111	color 15

PLAYFIELD 2Plane Bits Color Register

0000	color 0 (transparent, background color)
0001	color 17
0010	color 18
0011	color 19
0100	color 20
0101	color 21
0110	color 22
0111	color 23

AGA Depth From Here

1000	color 24
1001	color 25
1010	color 26
1011	color 27
1100	color 28
1101	color 29
1110	color 30
1111	color 31

The included example programs should be studied carefully for a demonstration of using dual playfield under AGA.

Bitplane usage by playfield:

Total Planes	Playfield 1	Playfield 2
2	0	1
3	0, 2	1
4	0,2	1,3
5	0,2,4	1,3
6	0,2,4	1,3,5
7	0, 2, 4, 6	1, 3, 5
8	0, 2, 4, 6	1, 3, 5, 7

Parallax Viewports

A "parallax viewport" (or pvp) is very much like a normal GameSmith `gs_viewport`, with the following differences:

- A pvp is installed with one of two new copper instructions.
- There are no blank lines between viewports. A new pvp can load in a single scan line.
- Pvp data automatically "wraps" when its boundaries are reached.

Your program is not given notice when this event occurs. For instance, if your program is scrolling a pvp to the right, and the new scroll value is past the right hand edge of the bitmap, then the bitmap pointers are reset to the left hand side. It's up to you to draw your graphics so that a smooth transition is effected.

Limitations

Currently, parallax viewports are limited to a depth of 3 bitplanes (there's only so much the copper hardware can accomplish between scan lines). However, plans are to have this supporting 4 bitplane (or more) reloads very soon. This may require smaller screen widths in some cases.

Actually, there is no software limitation here, but nothing over 3 bitplanes works properly yet with the hardware system.

Note: A color palette is NOT loaded automatically for a pvp. If you want to change the color palette as well as the display bitmap (very likely), you'll need to insert the proper color change copper instructions AFTER the UC_PVP instruction (see below and the included demo).

The pvp copper instruction

UC_PVP - install a parallax viewport.

The full instruction is as follows:

UC_PVP,y,&pvp_high,&pvp_low

y = scan line to wait on (relative to top of display window)

pvp_high = high order word of 32 bit address of `gs_pvp_struct`

pvp_low = low order word of 32 bit address of `gs_pvp_struct`

That's it! After filling a few fields in the pvp structure, this new copper instruction is all it takes to have the GameSmith display system automatically set up and handle a parallax field in your display.

The pvp structure

```
struct gs_pvp                                /* parallax viewport
{
struct gs_pvp *next;                        /* (SYSTEM USE ONLY!) next parallax viewport */
int height;                                /* height in scan lines */
int width;                                  /* width in pixels */
int depth;                                  /* depth of viewport */
int xoff;                                   /* X offset within bitmap */
int yoff;                                   /* Y offset */
unsigned long flags;
```

```

struct BitMap *bitmap[2];      /* ptr to bitmap page 1 & 2 */
/* SYSTEM FIELDS. DO NOT TOUCH! DANGER, HIGH VOLTAGE! */
unsigned long bplcon;          /* offset to horizontal shift load copper
                                instruction */
unsigned long bplptr;          /* offset to 1st bitplane pointer copper
                                instruction */
unsigned short scroll;          /* horizontal scroll value */
struct gs_viewport *vp;        /* ptr to gs_viewport this pvp belongs to */
unsigned char *LOF_planes1[8]; /* bitplane ptr temp hold for scroll reload
                                (lframe) */
unsigned char *SHF_planes1[8]; /* bitplane ptr temp hold for scroll reload
                                (sframe) */
unsigned char *LOF_planes2[8]; /* bitplane ptr temp hold for scroll reload
                                (lframe) */
unsigned char *SHF_planes2[8]; /* bitplane ptr temp hold for scroll reload
                                (sframe) */
};

```

height

This is the height of the pvp, and is the actual display height of the pvp, not any hidden bitmap amount which can be scrolled into view.

width

Width of the pvp in pixels. This is the total scrollable width. This number need NOT be any sort of even multiple of anything, but must be at least as wide as the parent viewport.

depth

Depth of the pvp in bitplanes. Currently, this should be 3 or less to work properly. Dual Playfield mode: in this mode, this should reflect the depth of the playfield being reloaded, not the actual depth of the entire viewport (i.e. in a 6 bitplane dual playfield display, a depth of 3 should be used to reload one of the playfields).

xoff / yoff

Don't worry about these fields. They are always initialized to 0 by the system before use.

flags

Valid pvp flags are:

PVP_DPF	Put the pvp in dual playfield mode
PVP_DPF1	Only load/scroll playfield 1
PVP_DPF2	Only load/scroll playfield 2
PVP_AGACOLOR	Used only by spvp, set AGA color regs. Unless this flag is set in the spvp structure, the old 12 bit type of color setup is assumed. Keep in mind that setting a 24 bit AGA color register takes 4 TIMES AS LONG as setting a 12 bit register.
PVP_RELOAD	SYSTEM FLAG: reload new scroll offsets

PVP_LINE1 spvp only: load color regs on the 1st line only of the spvp only. Allows a complete palette reload without a huge table encompassing changes on every scan line.

bitmap

These fields should contain pointers to the bitmaps you want displayed on pages 1 and 2 of your display. In a single buffered display, you can ignore the 2nd bitmap pointer, but it's a good idea to always initialize it with the same value as the 1st. Generally, both pages of a double buffered display will show the same pvp bitmap, and so it is perfectly legal to use the same bitmap pointer for both pages.

Sliced Parallax Viewport

A "sliced" parallax viewport (or spvp) is really nothing more than a high level construct which builds a pvp for every line of the bitmap. In this way, each line scrolls at a different rate, giving a real perspective feel. Scroll ratio from top to bottom is specified by means of two fields within the spvp structure.

Note that although a pvp can scroll both horizontally and vertically, a sliced parallax viewport is limited to scrolling in the horizontal direction.

This is an extremely powerful feature which slices a "full" bitmap automatically. However, there is quite a bit of processor overhead required to scroll such a beast on a per scan line basis. The included parallax demo is really pushing the limits of a stock 68000 at 7.14 MHz (however, it still runs at full frame rate). In addition, the scroll method provided not nearly as sophisticated as say "Lionheart". For this reason, future scroll methods will be available and easily selectable via the "scroll_type" field of the spvp structure.

The spvp copper instruction

UC_SPVP - install a sliced parallax viewport.

The full instruction is:

```
UC_SPVP,y,&spvp_high,&spvp_low

y = scan line to wait on (relative to top of display window)
spvp_high = high order word of 32 bit address of gs_spvp struct
spvp_low = low order word of 32 bit address of gs_spvp struct
```

Before the copper system can actually use an spvp structure, you must allocate the proper additional structures. You **MUST** do this before the UC_SPVP pseudo-op is processed by the GameSmith copper system. This is accomplished with the function `gs_alloc_spvp(spvp)`. Your program must free the extra memory used by an spvp before exiting, by calling the function `gs_free_spvp(spvp)`.

The spvp structure

```
struct gs_spvp /* sliced parallax viewport */
{
int top,bottom;           /* top and bottom movement values for scroll ratio
                           */
int height;               /* height in scan lines
int width;                 /* width in pixels */
int depth;                 /* depth of viewport */
unsigned long flags;       /* same as pvp */
struct BitMap *bitmap[2]; /* ptr to bitmap to slice (pages 1 & 2) */
unsigned long *ctable;     /* ptr to color table for color change each line */
```

```

unsigned short *cregs;          /* ptr to list of color reg to modify each line */
unsigned short nregs;          /* number of regs in list */
unsigned short scroll_type;     /* scroll type (0 through n), effects ratio setup &
                                scroller */
short inc,carry;               /*SYSTEM USE: Do Not Touch */
struct gs_pvp *list;           /*SYSTEM USE: Do Not Touch */
);

```

top

This is the scroll rate (in pixels) at which the top line of the spvp bitmap will scroll. Valid values range from 0 - n. Note that if using a top value of 0, the top line will actually move 1 pixel every so often, depending on the specified bottom value.

bottom

This is the scroll rate (in pixels) at which the bottom line of the spvp will scroll. With the only available scroll method, the bottom line will actually scroll at bottom or (bottom-1) with every scroll call.

height

This is the displayable height of the spvp. The specified value is the only thing the GDS system looks at when building the display, not the actual height of the bitmap being used.

width

Width of the spvp in pixels. This is the total scrollable width. This number need NOT be any sort of even multiple of anything, but must be at least as wide as the parent viewport.

depth

Depth of the spvp in bitplanes. Currently, this should be 3 or less to work properly. Dual Playfield mode: in this mode, this should reflect the depth of the playfield being reloaded, not the actual depth of the entire viewport (i.e. in a 6 bitplane dual playfield display, a depth of 3 should be used to reload one of the playfields).

flags

Valid pvp flags are:

PVP_DPF	Put the spvp in dual playfield mode
PVP_DPF1	Only load / scroll playfield 1
PVP_DPF2	Only load / scroll playfield 2
PVP_AGACOLOR	Used only by spvp, set AGA color regs. Unless this flag is set in the spvp structure, the old 12 bit type of color setup is assumed. Keep in mind that setting a 24 bit AGA color register takes 4 TIMES AS LONG as setting a 12 bit register.
PVP_RELOAD	SYSTEM FLAG: reload new scroll offsets
PVP_LINE1	Load color regs on the 1st line only of the spvp. Allows a complete palette reload without a huge table encompassing changes on every scan line.

bitmap

These fields should contain pointers to the bitmaps you want displayed on pages 1 and 2 of your display. In a single buffered display, you can ignore the 2nd bitmap pointer, but it's a good idea to always initialize it with the same value as the 1st. Generally, both pages of a double buffered display will show the same spvp bitmap. And so it is perfectly legal to use the same bitmap pointer for both pages.

ctable

This is a pointer to an array of color values to be effected on a per line basis (except when using the PVP_LINE1 flag, see above). Table entries are expected to be in the standard GameSmith color table format.

For instance, if you are changing 8 color registers per line, this table must be of size (8*height). Each bank of 8 colors are used in consecutive order every line.

cregs

This is a pointer to an array which contains the actual color numbers to modify each line (or only on the 1st line if using PVP_LINE1). This table may contain values from 0 - 31 for old type regs, and 0 - 255 for AGA registers (valid only with PVP_AGACOLOR). Entries can be in any order and of any valid number. For instance, if you are modifying 4 colors per line, you can list them in any way you choose, even something like 31,5,0,12. You need only setup your ctable properly to match the color regs being modified.

nregs

This is the number of color registers you will need modified per line (4 in the above example).

scroll_type

The only currently available scroll type is 0. In fact, this version of the software completely ignores this field. In the future, better perspective type scrolling will be provided as well, with the goal being to provide a "Lionheart" type of scroller in addition to this current one.

The current method looks like this: Imagine lying down on a flat board, and looking off to the side. Now imagine that a moving vehicle is dragging your board (either from your head or your feet). That's the angle and perspective shown by this method. Future methods would like to allow for different view angles such that they scroll in a realistic manner.

Drawing A Parallax Viewport so it Wraps Correctly

When designing your parallax field, it is important to remember, are:

- For proper horizontal wrapping, the left portion of the bitmap must be the same as the right portion. The width of the "portion" is defined by the width of the viewport in which the pvp resides. For instance, if I want to use a pvp bitmap in a viewport that is 320 pixels wide, the first 320 pixels of the bitmap must be the same as the last 320 pixels. Anything in between can be different. The "width" field of the pvp structure is used by the GDS display system to determine when it should horizontally wrap a pvp field.

- For proper vertical wrapping, the concept is the same as horizontal, except that instead of using the viewport width as a gauge, the height of the pvp field is substituted. The GDS display system uses the "Rows" fields of the bitmap structure itself to determine when a vertical wrap should take place. Note that this is only relevant to a pvp, not an spvp.

For an example of how all of this is put together, see the demo called "pdemo" on your example disks. It's really easier to implement than you may at first think and the technique can provide stunning effects.

Display Functions

<u>Function</u>	<u>Description</u>
gs_display	Simply create a low level display
gs_create_display	Create a low level display with all parameters
gs_show_display	Show the low level display
gs_flip_display	Show the other view in a double buffered display
gs_old_display	Show the display that was showing before creating a low level display
gs_remove_display	Release and remove a low level display
gs_LoadRGB	Reload entire color palette with new values
gs_SetRGB	Reload a single color palette entry with a new value
gs_scroll_vp	Scroll a viewport by delta X and Y
gs_scroll_vp_pf1	Scroll playfield 1 of a viewport
gs_scroll_vp_pf2	Scroll playfield 2 of a viewport
gs_scan_copper	Incorporate any changes to display copper list
gs_alloc_spvp	Allocate memory for a sliced parallax viewport
gs_free_spvp	Free memory for a sliced parallax viewport

5 The Animation System

Introduction

The GameSmith animation system is the most complete, programmable animation system available today. Its advanced, object oriented approach toward animated graphic objects frees the programmer from all but moving the object in a bitmap and handling collision detection. The interactive character animation tool CITAS allows you to graphically create your animation objects. All that your program needs to do is load or link with the object and simply reference it by the address of its controlling structure. That's it! In most cases, your program need not even be concerned with the size or depth of the object it is animating on the screen.

The animation system automatically and transparently handles:

- Double buffering
- Prioritized display of objects (objects can move in front of or behind other objects)
- Saving and restoring of background data "behind" objects
- Object-to-object collision detection between any number of collision detection areas
- Any number of object-to-background collision detection points so your program can determine the color of the background data behind your objects
- Translucent objects
- Animating an object forward or backward
- Single shot forward and/or backward animations (the animation system repeats the last cell until the sequence is reset)
- Automatic bitmap bounds checking, guaranteeing that your objects are never drawn outside of the bitmap (crash & burn)
- Automatic virtual space object coordinate handling
- Ability to shift real space within virtual space; perfect for tilemap scrollers
- Placement of attached, or child objects from the parent
- Automatic handling and placement of multi-sequence objects
- Automatic handling of objects in any number of simultaneous bitmaps

The following sections of this chapter will describe in detail:

- Controlling structures behind graphical objects
- Bitmap usage and display priorities
- The different types of objects
- How to make use of virtual and real space
- How to make use of bounds checking
- Object handling in single and double buffered environments
- Collision detection
- Object display methods and special effects
- Attached objects
- Disk based object files and objects that are linked with your program
- Building or loading an array of like objects
- Hardware Sprites

This section represents one of the core elements of the GameSmith Development System. The better your understanding of the concepts in this section, the greater your enjoyment of the system will be because you will be able to really utilize its power. We recommend several readings of this section. Do not try to absorb everything the first pass through.

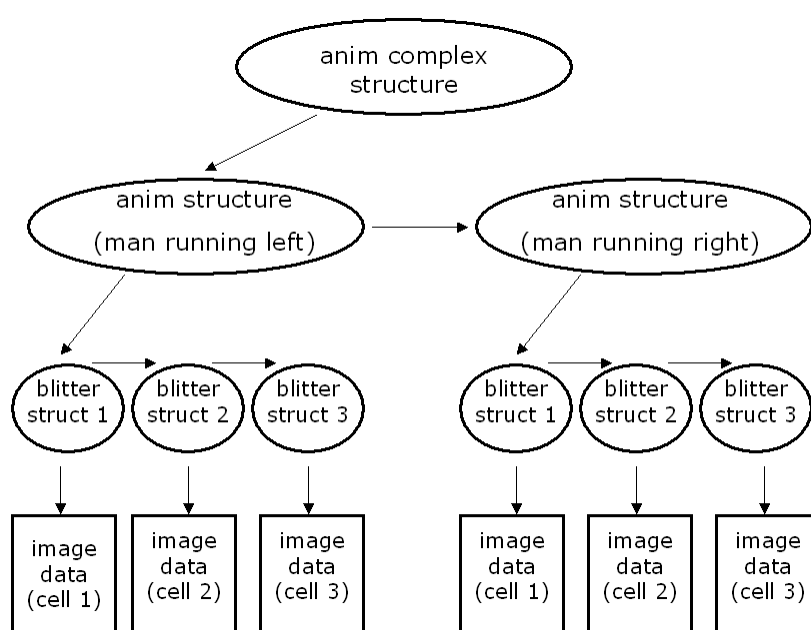
Structure Hierarchy

The animation system handles objects that range from fairly simple, to very complex. Even the simpler objects would be difficult and tedious to construct by hand, and the more complex objects VERY difficult and time consuming. Thus, the necessity for the utility CITAS.

The animated objects are constructed using a tiered hierarchy of controlling structures. The following chart shows an example of a complex anim (see the section on anim types for more information) with two separate animation sequences, each with three cells.

Note that the controlling structures that define collision detection areas and points are not shown in this chart. They are chained off of the blitter structures for each cell.

Animation System Structure Hierarchy



Bitmap Handling and Display Priority

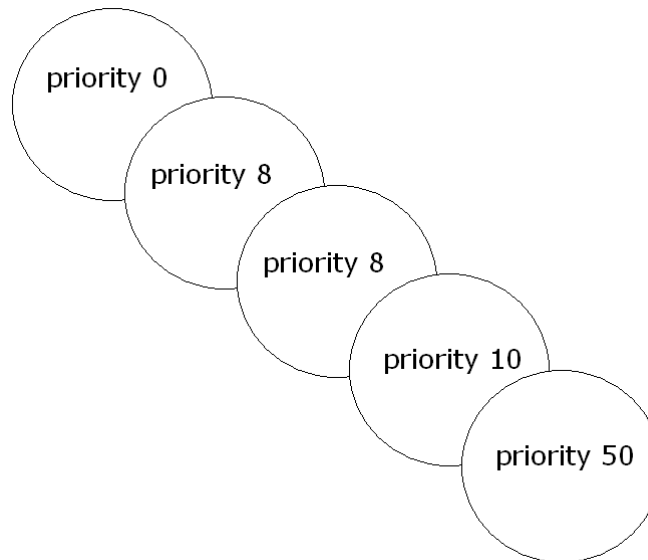
The animation system uses what are called "display lists" for object handling. Your program can allocate any number of display lists. Once you initialize a display list with a bitmap or bitmap pair (for single or double buffered display), objects added to the display list will automatically get drawn in the bitmaps you supplied.

Before an object can be controlled by the animation system, you must first add it to a display list. This is how the animation system knows about the object and where it should be drawn. You can add or remove objects at any time. It is important to note that objects in one list cannot interact with objects in another list; that is they cannot collide with each other. The ability to control multiple display lists has all sorts of uses, from multiple Intuition windows, to dual playfield mode, to split screen displays.

NOTE: an anim object must be present in only one display list at a time. Never try to add an object to multiple display lists.

When adding an object to a display list, it is inserted into the list according to its priority setting (defined by CITAS or set by your program). The priority defines the order in which the animation system will draw objects into the bitmap, thus allowing some objects to pass in front of or behind other objects.

The higher the priority, the higher the object is stacked on the screen (objects with a higher priority pass over the top of objects with a lower priority). Zero is the lowest possible priority. Objects with like priorities are simply drawn in the order in which they were added to the list, with the last object appearing in front of the previous, and so on. The anim priority field is a 16 bit unsigned value, which means object priorities may range from 0 to 65535 (a 64K range).



Display List Object Priorities

Object Types

Currently, there are only two types of animation objects: simple animation sequences, or simply "anims", and complex animations, or "anim complexes".

Simple Animation Objects

Simple animation objects, or anims, consist of a sequence of images, or cells, that, when displayed in rapid succession, appear to animate and move. This is the same technique employed in classical animations seen in Disney films or television cartoons. These are sometimes referred to as character animations, as many times the object is a character, or personage (like Snow White or Popeye). However, the animated object can just as well be any sort of object you want to give movement to, such as a leaf blowing in the wind, or Aladdin's magic carpet.

Normally, when you animate an object, the cells are displayed from the beginning to the end, or in a forward motion. You may opt to play back an animation in a reverse, or backward, motion.

Unless specified, animations will loop; meaning that after the last cell of the sequence is displayed, the animation is restarted at the first cell. A little planning will yield a smooth, continuous animation. Conversely, you may opt to create "single shot" animations, where the sequence does NOT loop.

You can specify single shot forward and/or backward animation. A single shot anim stops with the last cell, and you must reset it to a specific cell (generally cell 0) before animation will continue. The bubbles in the bubble example are a good example of single shot animations and their uses.

For the sake of controllability and ease of use, you may even create an anim that consists of only a single cell. This will allow you to use the image in the animation system, enabling very easy object handling and placement.

Complex Animation Objects

Complex animation objects, or anim complexes, consist of one or more simple animation sequences. Only a single cell of any given anim sequence is ever displayed at one time for the anim complex. To your program, the anim complex is one logical object. A good example might be an anim complex consisting of two animation sequences of a walking man; one walking left, and another walking right. Your program controls which anim sequence the complex will use by specifying the sequence number. Numbers can be confusing, however, and non-associative. CITAS provides logical names for these sequence numbers for you to make use of in your programs (such as MAN_LEFT and MAN_RIGHT). Use of these logical names makes your program much more readable.

Another very good example would be a character from the popular Street Fighter II video game. Each fighter has an arsenal of different moves with which to attack the opponent. Each move, or attack, consists of a unique animation sequence, like a jumping kick, or a punch. All of these moves make up a single character in the game. Just as the player sees them as one object, CITAS allows you to combine all necessary animation sequences into an anim complex, so that your program also sees them as one logical object, easily and simply controlled.

Your program can maintain the proper animation sequences on the screen through simple names such as LEFT_HIGH_PUNCH, or RIGHT_REVERSE_HOOK. Once you've set up the collision detection areas, the animation system takes care of the rest, notifying you when a flying foot smashes into the opponent's face, for instance.

Object Save Areas

In almost all cases, animated objects in a video game appear to move "on top of", or "in front of" the background scenery. They may also move in front of or behind each other. In actuality, what you see on the screen is the combination of all graphics data, background and anim objects, all represented in the same bitmap. In order to accomplish the 3D effect of moving, animated objects across a background, the animation system must save the background data "behind" each anim object on the screen.

This saved background data must be retained in another portion of memory known as a background data save area (referred to from now on as simply a save area). Every anim object can have up to two different save areas, depending on whether the object is being used in a single or double buffered display. Save areas are allocated automatically by the animation system, depending upon display type. The desired effect involves a 3 step process by the animation system:

- save background data where the image will go on the screen
- copy the image to the screen
- with next image movement, restore the background data behind the image (copy save area data back to the screen in the same position).

The process of allocating save areas usually takes place when you add the object to an animation display list, although you may opt to have the area(s) allocated beforehand. The animation system uses two methods of save area allocation, depending upon object type.

For simple anims, the animation system scans the sequence for the image that uses the most amount of memory (the largest and/or deepest cell). One or two save areas are then allocated at that size.

For complex anims, the animation systems scans every animation sequence in the complex for the image that uses the most amount of memory. One or two save areas are then allocated at that size. You can probably see that this method is a highly efficient way of dealing with animated objects moving across background graphics, especially in the case of complex anims (only one or two save areas are necessary for the entire object).

Note that save areas for objects that were NOT loaded from disk (i.e. linked with your program) must be manually freed (there are easy function calls for this).

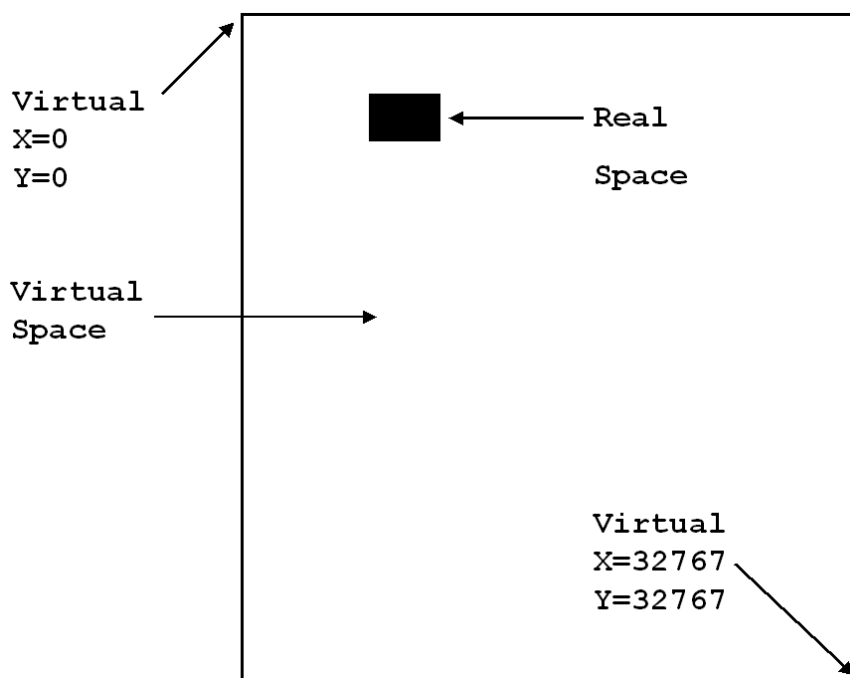
Virtual Space and Real Space

Object placement involves X and Y coordinates - standard Cartesian issue 2D space. The X value equates to pixels along the horizontal plane, with number values increasing as you move right. The Y value equates to pixels along the vertical plane, with number values increasing as down. i.e. coordinate 0,0 is the upper left hand corner.

One of the more powerful features of the animation system is the transparent handling of real space, inside of virtual space bounds. Object X and Y coordinates are 16 bit signed values. At this time, negative values are not allowed, and will produce undefined results. That leaves us with a 32K coordinate space in which objects are free to roam (from 0 to 32767 in both the X and Y directions).

This space is known as virtual space because obviously an Amiga could never make use of a real bitmap this large. Nonetheless, objects may be placed in this area, since the animation system will automatically draw objects when they cross into the bitmap, or real space.

The bitmap dimensions define real space, and may be placed at any offset within virtual space. Valid offset values range from 0,0 (real space lies in upper left hand corner of virtual space), to (32767 - bitmap_width), (32767 - bitmap_height) (bottom right hand corner of virtual space). Default real space offset for any given display list is 0,0. You may adjust these values with the function `gs_rs_offset()`.



Real Space – A Subset of Virtual Space

Bounds Checking

Bounds checking is the process of comparing the area of an object against a larger, bounded area. Default bounds for a given display list are the entire virtual space, but you may shrink this to some other value. If you shrink the valid object placement area to just the size of the bitmap, objects will never enter virtual space, and thus will always be visible in the bitmap. You may change object bounds at any time. The default for any given display list, is to do bounds checking on objects within the list. However, you may disable bounds checking if your program will handle all aspects of object placement. Disabling bounds checking may speed up your programs slightly.

If you allow the automatic bounds checking feature of the animation system, you can check objects for attempted boundary overlap after each object display update. There are four flag bits inside of the anim structure which are automatically set by the animation system in the event of a boundary overlap attempt (see the anim structure for a list of these bits and their meaning). You need not clear these bits yourself, as the animation system clears, and possibly resets them, with every display update (the `gs_draw_anims` function). The bits correspond to the left, top, right, and bottom boundaries accordingly. In the event that an object attempts to move, or be placed, outside of the boundaries, the appropriate bits are set and the object is placed at the outer edge of the boundary or boundaries.

Collision Detection

Collision detection is the process of determining if an object has collided with another object or with background data. You may separately enable or disable object-to-object and object-to-background collision detection for each individual object, or globally enable or disable object-to-object and object-to-background collision detection for a display list.

To globally enable collision handling, you give the animation system the address of the function you want to handle collisions. This function is called a collision "handler". The animation system can optionally call two different handlers per display list; one for object-to-object collisions, and one for object-to-background collisions.

Object-to-Object Collision Detection

Object-to-object collision areas take the form of invisible rectangles, of any size, which you place over the images in an anim sequence. CITAS allows you to graphically place, size, label, and type cast these areas. Every image, or cell, in the animation sequence gets its own set of collision detection areas, and you may set up any number of them for each image. Note that collision areas cannot collide with each other in the same object, only with areas of other objects.

You can define up to 32 different types of collision detection areas to be used at any given time in the animation system. Some examples of collision area types might be labeled HEAD, BELLY, FOOT, SWORD, and SHIELD. You can then set up a mask which defines which types are allowed to collide with each other. For instance, a SWORD type would probably collide with all other types, but a FOOT would probably not collide with another FOOT. The animation system will call your object-to-object collision handler when two collision enabled areas overlap, passing the addresses of the colliding objects, and the two areas that are overlapping.

```
void collision(anim1,anim2,coll1,coll2)
struct anim_struct *anim1;
struct anim_struct *anim2;
struct collision_struct *coll1;
struct collision_struct *coll2;
```


Always remember that the animation system only sees 32 different area types, regardless of how you refer to them. This means that to the animation system, area 1 will always be area 1 no matter if you call it a HEAD or a FOOT. CITAS allows you to set up any number of different collision tables (a full set of 32 definitions). You should make sure that all of the objects that can possibly be on the screen at the same time use the same collision table. Your program might want to refer to a different collision table, for instance, with each new level of a game that requires totally different objects.

Being able to set up multiple collision tables is a powerful and flexible tool, however it can lead to confusion if you do not carefully keep track of your object. In most cases, you will be able to set up one collision table that can be used throughout the entire game. This will avoid any possible confusion. If you feel uncomfortable with the notion of multiple collision tables, stick to one for a while.

When defining object-to-object collision detection areas, try to restrict the number of areas to only those that are truly necessary. Depending on processor speed, a large number of collision detection areas will greatly decrease performance. The number of area overlap comparisons which must be made every update is equal to $(n * (n-1))/2$, where n is the number of collision detection areas. This model assumes one collision detection area per object. That number decreases slightly when using multiple areas per object.

NOTE: By default, object-to-object collisions only happen in real space. If you want your objects to be able to collide in virtual space, set the ANIM_VCOLLIDE flag bit in the anim structure (see the section describing the anim structure).

Object-to-Background Collision Detection

Object-to-background collision detection allows your program to determine when objects overlap certain background colors. For instance, you may want to determine when your character is standing on water instead of grass. Instead of the rectangular areas that allow object-to-object collision detection, object-to-background collision detection is done pixel by pixel. You define the points/pixels on each image where you want the animation system to automatically check the background data for a collision. Again, CITAS allows you to graphically place, label, and mask each point. The point mask allows you to select the bitplanes that can cause a collision. In this way, you can exclude certain color combinations from causing a collision.

The animation system calls the function `gs_plot_test` to determine the color of the background data behind each point. When a valid collision takes place, the animation system calls your object-to-background collision handler with the address of the anim object, the address of the background collision point, and the color value of the background data.

```
void collision_bg(anim, collision, color)
struct anim_struct *anim;
struct coll_bg_struct *collision;
int color;
```

Note that background collision detection cannot detect collisions with other objects. Only the non-object data that is already in the bitmap(s) BEHIND anim objects is tested. Obviously, background collisions cannot take place in virtual space, either.

As with object-to-object collision detection, try not to over use the background points. Setting up too many background collision points will noticeably slow down your program. Remember that the animation system must call `gs_plot_test` for every point on every visible image in the animation system with every display update. With a little practice, you'll learn how to be efficient with point selection.

Display Methods

Anim objects can be copied to a bitmap in one of three distinct ways, each graphically selectable via CITAS (pull up the Anim Options window).

Save/Restore Background

This is the default display method, and allows objects to move over a background scene without disturbing it (see the discussion of save areas). Each time an object moves or animates:

- the background data in the previous location is restored
- the background data behind the object at the new location is saved,
- the object image is copied to the bitmap.

This is the slowest method of the three because it involves 3 steps with every movement. However, it is the only one that allows background data to remain undisturbed during object movement.

Erase Image

This method does NOT save and restore background data. Instead, with every movement or animation of an object, 1) the image is erased from the bitmap using the blitter mask (the area where the image was located is cleared to zero, or background color), and 2) the image is copied (using the blitter mask) to the new location. This method can only effectively be used in a display without background data (a space scene?).

Simple Copy

This is the fastest, simplest method. Unlike the other two display methods, the blitter mask is not used, and the entire rectangle of image data is merely copied to the destination bitmap. Moving an object of this type in a bitmap will leave a trail, or copy, of itself as it moves.

Special Effects

At this time, there are really only two special effects which can be applied to an object regardless of display type. The first is referred to as the "flicker effect", and the second is translucency.

Object Flicker

This effect can only be achieved in a double buffered display. When turned on, the object will appear to "flicker", or blink, on the screen. This is the result of the animation system only displaying the object in the first bitmap. The rate at which the object flickers is determined by how fast you flip display pages. A fairly simple technique, but widely popular in games.

Typically, a player character will flicker when a new life is used, and during this time the player cannot be damaged. This is easily achieved by setting flicker mode and disabling collision detection for the object. After an allotted time period, allowing the player to get his or her bearings, you would turn off flicker and re-enable collision detection.

Translucency

A translucent object allows light to pass through it to one degree or another. For instance, a window, colored or otherwise, can be seen through, and objects behind the window are visible. A translucency effect can be achieved by merging image data with background data (or overlapping objects). Normally, when an object is copied to a bitmap, the bitplanes behind an image which are unaffected are cleared to zero. This guarantees that the image will appear with the proper colors.

For instance, if an object is comprised of 2 bitplanes (4 colors), using planes 0 and 1, and is copied to a bitmap with a depth of 4 bitplanes, the image area in bitplanes 2 and 3 are cleared to zero when the image is copied to the bitmap. When translucency is turned on, however, this is not the case, and bitplanes 2 and 3 would remain unchanged during the copy operation.

A properly designed color palette can yield fascinating results in translucency mode. CITAS allows you to set translucency mode for an anim object. This sets the ANIM_MERGE bit in the anim structure, which can also be set or cleared through function calls.

Attached Anims

The animation system is capable of handling objects that are chained together. In this scenario, the first object is referred to as the "parent", and all attached objects as the "children" of the parent. Placement of the child anims is automatic, and relative to the position of the parent object. Therefore, moving the parent object also moves the children. This is useful, for instance, if you have a large object with only a few moving parts.

Attaching the animated parts to the large image greatly reduces the amount of RAM required to make up the entire object. A good example might be a dragon that has only the head, neck, and feet as moving parts; the rest of the body is a still image. You might also attach an animated flame to the dragon's mouth at certain times. Your choice of a parent object is arbitrary, but should be the object you would want to refer to as the positioning point for the entire object chain (such as the dragon's head).

At this time, CITAS does not allow you to graphically attach anim objects. This capability will be provided in a future release.

To attach an object to another, follow these steps:

- Set the ANIM_PARENT bit in the anim structure of the chosen parent object.
- Fill the "attach" field in the anim structure of the parent object so that it points to the anim structure of the first child object.
- Set the "xa" and "ya" fields in the anim structure of the child object to the desired offsets. Remember that object placement is relative to the upper left hand corner of the object. For instance, an xa offset of -20 would always place the child object 20 pixels to the left of the parent's position.

Multiple objects may be chained by filling the attach field and xa and ya offsets for the additional children. Note that the xa and ya fields always refer to an offset from the parent object position, not the previous object in the chain. The last object in the chain MUST contain a NULL pointer in the attach field. This marks the end of the chain. Note that only the parent object should have the ANIM_PARENT flag bit set.

After objects have been linked together, it is still possible to individually place and move the child objects. Be aware, however, that moving or animating the parent object will place all of the children relative to the parent.

Disk Based Anim Objects

One of the most powerful features of CITAS is the ability to create disk based animation objects. The disk based anim file contains all information about the object, so that the load function can duplicate the object in its entirety. In fact, your program can load an object, of any type, with a single function call. The object can then be added to an animation display list and used directly. Your program need know virtually nothing about the object itself. This means that the graphic objects and the code are truly separate, and development of each can remain apart. Beyond that, it opens up all kinds of interesting ideas for object interchangeability, object additions, and object enhancements without ever changing the program code.

Disk based objects may be loaded and freed as needed, using only the amount of RAM necessary at any given time (remember to release disk based objects by calling the appropriate free function). This goes a long way towards the construction of very large games. In addition, disk based objects may be arbitrarily cloned at load time (see the section on object arrays). This is something you simply cannot do with linked objects.

It can be argued that disk based objects are vulnerable to manipulation by the end user or the pirate. This will always be true to some extent, however Oregon Research has taken steps against it. Not only is the graphic data itself encrypted and compressed within the final version of anim files, but they may also be "locked" to your serial number. This means that ONLY your program can load your objects, and even others with a copy of the GameSmith Development System won't be able to use your objects in their programs. In addition, the final output version (see Chapter 6 CITAS) of anim files cannot be reloaded into CITAS. So in many respects, the disk based objects are even more secure than objects that are linked into your program.

If you allowed a color table to be allocated at load time, it must be freed by you. Use the FreeMem Exec function. Example:

```
FreeMem(load.cmap, load.cmap_entries*sizeof(long));
```

Where `load.cmap` is the address of the color table (from the `anim_load` structure), and the second parameter is the size (in bytes) of the memory block you are freeing (long must, be a 32 bit integer, or 4 bytes). For more information on the FreeMem function, refer to the AMIGA ROM Kernel Reference Manual: LIBRARIES.

Linked Anim Objects

The alternative to loading disk based anim objects at run time is to generate source code for them using CITAS. You then simply compile or assemble the source output and link them into your program. This way the objects get loaded automatically when your program is started.

There are advantages and disadvantages to linked anim objects. The advantages are that your game can be distributed entirely as a single file, objects get loaded automatically when your program is started, and that the objects themselves are hidden from the user. The disadvantages are that you cannot free up the memory used by these objects, development time is longer due to additional compile (or assemble) and link times, objects cannot be arbitrarily cloned (the array size is fixed), and you cannot interchange, modify, or add graphic objects after the program is compiled and linked.

You don't need to free linked objects, but you do need to make sure that any save areas for the objects are deallocated before your program exits. Use one of the appropriate "free save areas" functions.

Object Arrays

Whether you are using linked anim objects or disk based anim objects, you can always refer to them as an array. This holds true for any anim type. For linked anims, the array number is fixed, and defined in the like named header or include file (see the CITAS manual for more information). For disk based anims, your program specifies how many elements you would like in the array, and the loader does the rest.

The anim array, whether a simple anim or a complex anim, may contain any number of elements (memory allowing) except 0. If the array contains more than one element, each object is an exact duplicate except for the array num field, which contains the element number (from 0 to n). The array element number is useful in such cases as collision detection. Note that there is only one copy of the image data itself, as well as some structures. This greatly conserves memory. However, the animation system will allocate a save area for each element if so required by the object's display method (when the object is added to a display list).

The ability to create multiple element arrays of the same object can be an extremely useful and efficient feature. Any time your program requires use of more than one copy of the same anim on screen at the same time, you should allocate the objects as a multiple element array.

Objects in Dual Playfield Mode

There are a few considerations when using anim objects in dual playfield mode. As you know, in dual playfield mode (see Chapter 4, The Display System) each playfield uses every other bitplane in the bitmap. Instead of creating a special anim type for dual playfield mode, you'll need to initialize your display list in a slightly different way. Follow the procedure below in order to use anim objects in a single playfield of a dual playfield display.

This example assumes a double buffered display. We will be adding anim objects to playfield 1 only, but this same method can also be used for the 2nd playfield.

- Define two BitMap structures BESIDES those used for the actual display.
- Copy all fields except the Planes array of the BitMap structures used for the actual display to your extra BitMap structures.
- For playfield 1, copy the display Planes 0, 2, 4, and 6 to extra BitMap Planes 0,1,2,3. Repeat for 2nd BitMap.
- Initialize the animation display list with your extra BitMap copies. That's it, now the animation system will correctly draw your anim objects in playfield 1!

The 'C' code to handle this would be:

```
void init_anim_pf1(struct BitMap *bm1, struct BitMap *bm2, int dlist)
{
    static struct BitMap pfla,pflb;
    pfla.BytesPerRow = bm1->BytesPerRow;
    pflb.BytesPerRow = bm2->BytesPerRow;
    pfla.Rows = bm1->Rows;    pflb.Rows = bm2->Rows;
    pfla.Flags = bm1->Flags;  pflb.Flags = bm2->Flags;
    pfla.Depth = bm1->Depth;  pflb.Depth = bm2->Depth;
    pfla.Planes[0] = bm1->Planes[0];
    pflb.Planes[0] = bm2->Planes[0];
```

```

pfla.Planes[1] = bm1->Planes[2];
pflb.Planes[1] = bm2->Planes[2];
pfla.Planes[2] = bm1->Planes[4];
pflb.Planes[2] = bm2->Planes[4];
pfla.Planes[3] = bm1->Planes[6];
pflb.Planes[3] = bm2->Planes[6];
gs_init_anim(dlist, &pfla, &pflb);
};

```

The Anim Structures

Anim objects are made up of a complex web of controlling structures. At different times throughout your program, you will need to refer to some of the fields within at least the higher level structures. This section details all of the structures used by the animation system. (The blitter structure is not detailed here as it is considered part of the general purpose graphics routines. See the Chapter 3 The Graphics System for information on the blitter structure.)

The Background Collision Structure

The background collision structure defines a single object-to-background collision detection point on an image/cell.

```

struct coll_bg_struct
{
    unsigned int mask;
    unsigned int area;
    unsigned short x1;
    unsigned short y1;
    struct coll_bg_struct *next;
};

```

mask

This is the collision enable mask. It defines the bitplanes with which this point can collide. A 1 bit indicates that this point can collide with the given bitplane. For example, a mask of 0x0f indicates that the point can collide with any or all of the first 4 bitplanes in the bitmap.

area

This field contains the collision area number. Collision areas (or points) are numbered beginning with 0 for the first one, and progress incrementally through the last one in the chain for each image. Using CITAS, you may optionally define logical names for each area. If your program needs to refer to specific areas within an image, then it is handy to reference them by name instead of number. Referral by name will guarantee that your program still refers to the correct point even after modifying the object-to-background collision points (you will need to recompile or reassemble your program).

x1

This is the X coordinate offset, from the upper left hand corner of the image, which defines the position of the collision point.

y1

This is the Y coordinate offset, from the upper left hand corner of the image, which defines the position of the collision point.

next

This is a pointer to the next object-to-background collision detection point for the image. This field contains NULL for the last point.

The Object Collision Structure

The object collision structure defines a single object-to-object collision detection rectangle for an image/cell.

```
struct collision_struct
{
    unsigned int type;
    unsigned int mask;
    unsigned int area;
    unsigned short x1;
    unsigned short y1;
    unsigned short x2;
    unsigned short y2;
    struct collision_struct *next;
};
```

type

This is the area type, and will contain a single true bit (one and only one of the 32 bits will be set). Normally, your program will refer to the type by name (such as HEAD, FOOT, SWORD, etc.).

mask

This is the collision enable mask. This determines what other types this area can collide with. Each collision enabled type will have its corresponding bit set in the mask. For instance, if the area can collide with types 1 and 32, the mask would contain the hex value 0x80000001.

area

This field contains the collision area number. Collision areas are numbered beginning with 0 for the first one, and progress incrementally through the last one in the chain for each image. Using CITAS, you may optionally define logical names for each area. If your program needs to refer to specific areas within an image, then it is handy to reference them by name instead of number. Referral by name will guarantee that your program still refers to the correct area even after modifying the object-to-object collision areas (you will need to recompile or reassemble your program).

x1

This is the X coordinate offset, from the upper left hand corner of the image, which defines the left edge of the collision rectangle.

y1

This is the Y coordinate offset, from the upper left hand corner of the image, which defines the top edge of the collision rectangle.

x2

This is the X coordinate offset, from the upper left hand corner of the image, which defines the right edge of the collision rectangle.

y2

This is the Y coordinate offset, from the upper left hand corner of the image, which defines the bottom edge of the collision rectangle.

next

This is a pointer to the next object-to-object collision detection area for the image. This field contains NULL for the last point.

The Anim Structure

The anim structure defines an entire chain of images, or one animation sequence, which are in turn controlled by blitter structures. This is the controlling structure for a simple anim object.

```
struct anim_struct
{
    struct blit_struct *list;
    struct blit_struct *img;
    struct anim_struct *attach;
    unsigned short array_num;
    unsigned short label;
    unsigned short count;
    short x;
    short y;
    short xa;
    short ya;
    unsigned short width;
    unsigned short height;
    unsigned short cell;
    unsigned short prio;
    unsigned long flags;
    /*****internal use only*****/
    /* Except for collide, cplx_next, and cplx, */
    /* your program should never directly access the following fields */
    unsigned short shift1;
    unsigned short shift2;
    void *save1;
    void *save2;
    unsigned int offset1;
    unsigned int offset2;
    unsigned int savsiz;
    struct blit_struct *image1;
    struct blit_struct *image2;
    struct anim_struct *collide;
    struct anim_struct *prev;
    struct anim_struct *next;
    struct anim_struct *cplx_next;
    struct anim_cplx *cplx;
};
```


list

This is a pointer to the first blitter structure in the sequence. The blitter structure in turn controls the image data itself. The blitter structures are chained together in a linked list, forming one animation sequence.

img

This is a pointer to the current image on the screen. This pointer is only valid after the anim has been added to a display list.

attach

This is a pointer to the next attached anim. If there are no more attached anims, then this field must be NULL.

array_num

This is the object's array element number, from 0 to n.

label

This is the user defined label/ID field. Objects created with CITAS will have a unique label.

count

This field contains the number of images in the animation sequence.

x

Current X coordinate of the object.

y

Current Y coordinate of the object.

xa

This is the X offset from the parent's position at which this anim will be placed. See the section on attached anims for more details. This applies only to child, or attached, anims.

ya

This is the Y offset from the parent's position at which this anim will be placed. See the section on attached anims for more details. This applies only to child, or attached, anims.

width

This is the maximum width of the anim in bytes. This field is equal to the widest image in the sequence. Note that the animation system automatically updates this field when the object is added to a display list.

height

This is a maximum height of the anim in rows. This field is equal to the tallest image in the sequence. Note that the animation system automatically updates this field when the object is added to a display list.

cell

This contains the current cell number being displayed (from 0 to n). This field is only valid after the anim is added to a display list.

prio

This is the anim's display priority. See the section on display priority for more details. You may change this field at any time, and then call `gs_sort_anims` to reprioritize the list.

flags

The user anim flags are as follows:

ANIM_SAVE_BG	Use the Save/Restore Background display method.
ANIM_COLLISION	Object-to-object collision detection enabled. Never set or clear this bit directly. Use the enable/disable function calls instead.
ANIM_ACTIVE	Anim is active in the animation system (it is part of a display list).
ANIM_INVISIBLE	Anim is part of a display list but is invisible on screen. Never set or clear this bit directly. Use the clear/enable functions instead.
ANIM_ONSCREEN	Anim is displayed in a bitmap. READ ONLY!
ANIM_CLEAR	Use the Erase Image display method.
ANIM_COPY	Use the Simple Copy display method.
ANIM_REVERSE	Object is animating in reverse. Never set or clear this bit directly. Use the forward/reverse functions instead.
ANIM_PARENT	Anim is a parent with attached anims.
ANIM_MERGE	Merge image data with background data to cause a translucency effect. Never set or clear this bit directly. Use the <code>set_anim_merge/clear_anim_merge</code> functions instead.
ANIM_FLICKER	Display object only in the first bitmap. Never set or clear this bit directly. Use the <code>set_anim_flicker/clear_anim_flicker</code> functions instead.
ANIM_COLLISION_BG	Object-to-background collision detection enabled. Never set or clear this bit directly. Use the enable/disable functions instead. Used ONLY by the animation system. It indicates that the anim uses a copy of the image data. DON'T TOUCH!

ANIM_BOUNDS_X1	The anim object tried to move past the left edge of the bounds. Read only.
ANIM_BOUNDS_Y1	The anim object tried to move past the top edge of the bounds. Read only.
ANIM_BOUNDS_X2	The anim object tried to move past the right edge of the bounds. Read only.
ANIM_BOUNDS_Y2	The anim object tried to move past the bottom edge of the bounds. Read only.
ANIM_CPUBLIT	Use the CPU to display the object instead of the Amiga's blitter chip.
ANIM_VSPACE	Object lies in virtual space. READ ONLY!
ANIM_VCOLLIDE	Set this bit if you want to allow objects to collide with each other in virtual space.
SPECIAL NOTE:	Only one of ANIM_SAVE_BG, ANIM_CLEAR, or ANIM_COPY should ever be set at any time, but one MUST be set.

Shift1

Save area for blitter shift value, bitmap 1. Used for optimized restore/ erase.

shift2

Save area for blitter shift value, bitmap 2. Used for optimized restore/ erase.

save1

Pointer to the background data save area used for bitmap 1.

save2

Pointer to the background data save area used for bitmap 2.

offset1

Bitmap offset for save area 1. Used for optimized restore.

offset2

Bitmap offset for save area 2. Used for optimized restore.

savsiz

The size of the save area in bytes.

image1

Pointer to the image displayed in bitmap 1.

image2

Pointer to the image displayed in bitmap 2.

collide

Pointer to the colliding anim.

prev

Pointer to the previous anim in the display list.

next

Pointer to the next anim in the display list.

cplx_next

Pointer to the next anim in an anim complex.

cplx

Pointer to the anim complex to which this anim belongs.

The Anim Complex Structure

The anim complex structure is the controlling structure for the complex anim object type. It controls one or more simple anims, or animation sequences.

```
struct anim_cplx
{
    unsigned short cnt;
    unsigned short seq;
    unsigned short width;
    unsigned short height;
    unsigned short array_num;
    unsigned short label;
    struct anim_struct *list;
    struct anim_struct *anim;
};
```

cnt

This contains the number of anims within the complex.

seq

This contains the current animation sequence (from 0 to n).

width

This is the maximum width of the complex in bytes. This field is equal to the widest image from all sequences. Note that the animation system updates this field when the complex is added to a display list.

height

This is the maximum height of the complex in rows. This field is equal to the tallest image from all sequences. Note that the animation system updates this field when the complex is added to a display list.

array_num

This is the object's array element number, from 0 to n.

label

This is the user defined label/ID field. Objects created with CITAS will have a unique label.

list

Pointer to the first animation sequence in the complex. Anims are chained together in a linked list.

anim

Pointer to the current anim in use. This field is only valid after the complex has been added to a display list.

The Anim Load Structure

The anim load structure is your interface to the anim loader routines which load disk based anims into memory. The loader can load editable anim files, final version anim files, and locked final version anim files.

```
struct anim_load_struct {
    char *filename;
    union {
        struct anim_struct *anim;
        struct anim cplx *cplx;
    }
    anim_ptr;
    struct anim_attach *attach;
    unsigned long *cmap;
    int cmap_entries;
    int cmap_size;
    int type;
    int array_elements;
    unsigned long flags;
};
```

filename

This is a pointer to a null terminated string containing the name of the file to load. Full AmigaDOS path names are legal.

anim

This is a pointer to the object's controlling structure upon successful load. If the anim type is a simple anim, you should reference it through this pointer. Note that this field and the cplx field share the same space in memory, and point to the same address.

cplx

This is a pointer to the object's controlling structure upon successful load. If the anim type is a complex anim, you should reference it through this pointer. Note that this field and the anim field share the same space in memory, and point to the same address.

attach

This is a pointer to the attachment array specification, which remains undefined at this point. This field will contain a NULL pointer if there is no attachment array specification.

cmap

This is a pointer to a color table if you allowed the load routines to allocate one, and the load operation was successful.

cmap_entries

Upon successful completion of a load and color table allocation, this field will contain the number of entries in the color table.

cmap_size

This field must be set to the number of desired bits per color value when loading a color table (red, green, blue respectively). Current value lengths are 4 and 8. Make sure to set this field before calling the loader.

type

This field contains the object type upon successful completion of the load. Currently, the type definitions are:

- 0 simple anim
- 1 complex anim

array_elements

This must be set to the number of desired elements in the array. This value must be equal to 1 or greater, and cannot be negative. Make sure to set this field before calling the loader.

flags

The loader flags are as follows:

ANIMLOAD_NOCOLOR	Do not allocate a color table
ANIMLOAD_FASTRAM	Override any object settings in the file and try to load it to Fast RAM. If the object is successfully loaded, the loader will set the CPUBLIT flags for this object. The object will be displayed using the Amiga's CPU.
ANIMLOAD_NOFASTRAM	Override any object settings in the file and try to load it to Chip RAM. If the object is successfully loaded, the loader will clear any CPUBLIT flags for this object. The object will be displayed using the Amiga's blitter chip.

Hardware Sprites

The GameSmith display system allows use of hardware sprites. Sprite usage is limited to 8 individual sprites, or 4 attached sprites. AGA sprite widths are not supported yet (32 or 64 bits in width).

Sprite data is exactly that described in the Amiga Hardware Reference Manual. Make sure that the sprite data bank has two blank words at the beginning, and two NULL words (16 bits each) at the end. Hardware sprites are manipulated with the system functions `gs_add_sprite()`, `gs_move_sprite()`, and `gs_clear_sprite()`.

Refer to the example code on the disks for examples of the use of hardware sprites.

Attached Sprites

As per the reference manual, sprites may be attached in the following manner:

sprite 1 to sprite 0

sprite 3 to sprite 2

sprite 5 to sprite 4

sprite 7 to sprite 6

To attach a sprite (thus doubling it's color palette), simply set bit 7 in the 2nd word of the sprite data for the odd numbered sprite (1,3, 5, or 7). If you do NOT want the sprite to be attached, make sure bit 7 is off (zero).

The following is an example of the first two words in an attached sprite data bank:

```
0x0000,0x0080, /* note, bit 7 is set */
```

Or, you can set this bit in your program with the following 'C' statement:

```
sprite[1] |= 0x80; /* set the attach bit */
```

Remember to keep your attached sprites in the same X & Y position.

Animation Functions

<u>Function</u>	<u>Description</u>
<code>gs_load_anim</code>	Load an anim object from disk
<code>gs_free_anim</code>	Free an anim object from memory
<code>gs_free_cplx</code>	Free a complex anim object from memory
<code>gs_get_display_list</code>	Allocate a display list
<code>gs_free_display_list</code>	Free a display list
<code>gs_init_anim</code>	Initialize a display list for use
<code>gs_rs_offset</code>	Specify real space X and Y offsets within virtual space
<code>gs_rs_window</code>	Shift a real space window which is smaller than the bitmap area to any position over the bitmap
<code>gs_set_anim_bounds</code>	Specify the legal bounds within which anim objects may roam
<code>gs_disable_anim_bounds</code>	Turn off bounds checking

<code>gs_enable_anim_bounds</code>	Turn on bounds checking
<code>gs_next_anim_page</code>	Tell the animation system to draw into the other bitmap of a double buffered display
<code>gs_get_anim_save_area</code>	Allocate memory for saving background data
<code>gs_get_parent_save_area</code>	Allocate background save memory for a parent object and all children
<code>gs_free_anim_save_area</code>	Release save area memory
<code>gs_free_parent_save_area</code>	Release save area memory for a parent object and all attached children
<code>gs_add_anim</code>	Add an anim object to a display list
<code>gs_add_parent</code>	Add a parent anim and all children to a display list
<code>gs_remove_anim</code>	Remove an anim object from the animation system
<code>gs_sort_anims</code>	Sort all anim objects in a display list
<code>gs_draw_anims</code>	Copy all anim objects in both display lists to the current drawing bitmap
<code>gs_restore_anim_bg</code>	Remove all anim objects from the current drawing bitmap
<code>gs_draw_anim_objs</code>	Copy all anim objects in both display lists to the current drawing bitmap after first calling <code>gs_restore_anim_bg</code>
<code>gs_reverse_anim</code>	Reverse an object's animation direction
<code>gs_anim_forward</code>	Set an object's animation direction to forward
<code>gs_anim_backward</code>	Set an object's animation direction to backward
<code>gs_clear_anim</code>	Make an active anim object invisible
<code>gs_enable_anim</code>	Make an invisible anim object visible again
<code>gs_set_anim_flicker</code>	Cause an anim object to flicker when displayed
<code>gs_clear_anim_flicker</code>	Stop an anim object from flickering
<code>gs_set_anim_merge</code>	Turn on translucency for an anim object
<code>gs_clear_anim_merge</code>	Turn off translucency for an anim object
<code>gs_enable_anim_collision</code>	Allow an anim object to collide with other anim objects
<code>gs_disable_anim_collision</code>	Disallow an anim object to collide with other anim objects
<code>gs_enable_anim_collision_bg</code>	Allow an anim object to collide with background data
<code>gs_disable_anim_collision_bg</code>	Disallow an anim object to collide with background data
<code>gs_set_anim_cell</code>	Set an object's current animation frame
<code>gs_move_anim</code>	Move an anim object to an X,Y coordinate in a bitmap
<code>gs_anim_obj</code>	Move and animate an anim object
<code>gs_anim_parent</code>	Move and animate a parent anim and all attached children
<code>gs_anim_children</code>	Move a parent anim, but only animate the child anims
<code>gs_set_collision</code>	Supply a collision handler for object-to-object collisions from a display list
<code>gs_clear_collision</code>	Disable collision handling for object-to-object collisions from a display list
<code>gs_set_collision_bg</code>	Supply a collision handler for object-to-background collisions from a display list
<code>gs_clear_collision_bg</code>	Disable collision handling for object-to-background collisions from a display list
<code>gs_add_anim_cplx</code>	Add a complex anim object to a display list
<code>gs_remove_cplx</code>	Remove an anim complex from the animation system
<code>gs_get_cplx_save_area</code>	Allocate a save area for background graphics
<code>gs_free_cplx_save_area</code>	Release save area memory
<code>gs_set_cplx_prio</code>	Set the display priority for an anim complex
<code>gs_set_cplx_seq</code>	Set the current animation sequence for an anim complex

<code>gs_set_cplx_cell</code>	Set the current animation frame for the current anim sequence of an anim complex
<code>gs_set_cplx_flicker</code>	Cause an anim complex to flicker when displayed
<code>gs_clear_cplx_flicker</code>	Stop an anim complex from flickering
<code>gs_set_cplx_merge</code>	Turn on translucency for an anim complex
<code>gs_clear_cplx_merge</code>	Turn off translucency for an anim complex
<code>gs_clear_cplx</code>	Make an active anim complex invisible
<code>gs_enable_cplx</code>	Make an invisible anim complex visible again
<code>gs_reverse_cplx</code>	Reverse an anim complex's animation direction
<code>gs_cplx_forward</code>	Set an anim complex's animation direction to forward
<code>gs_cplx_backward</code>	Set an anim complex's animation direction to backward
<code>gs_enable_cplx_collision</code>	Allow an anim complex to collide with other objects in the same display list
<code>gs_disable_cplx_collision</code>	Disable an anim complex from colliding with other objects in the same display list
<code>gs_enable_cplx_collision_bg</code>	Allow an anim complex to collide with background data
<code>gs_disable_cplx_collision_bg</code>	Disable an anim complex from colliding with background data
<code>gs_move_cplx</code>	Move an anim complex to an X,Y coordinate in a bitmap
<code>gs_anim_cplx</code>	Move and animate the current animation sequence of an anim complex
<code>gs_anim_cplx_parent</code>	Move and animate the current animation sequence of an anim complex and all children
<code>gs_anim_cplx_children</code>	Move an anim complex, but only animate attached child anims
<code>gs_re_dim</code>	Specify a smaller real space than the entire bitmap area
<code>gs_add_sprite</code>	Instructs the display system to use your sprite data for the specified sprite number
<code>gs_move_sprite</code>	Moves a sprite to specified position
<code>gs_clear_sprite</code>	Make sprite invisible

6 CITAS - Object Animation Tool

One of the largest stumbling blocks for programmers is creating and controlling animated objects in a game. Before the advent of CITAS, combining and aligning the different images of a character animation was quite tedious and time consuming, to say nothing of setting up collision detection for the object. CITAS is a utility which provides an easy graphical way to construct anim objects - from the image level all the way through to collision detection. You even set up the anim display method using CITAS. In short, everything about the object is set up through CITAS (although your program is by no means restricted to this setup information). When finished, your anim objects can be used directly in the GameSmith animation system. It's simple and easy.

Image Creation

You may use any paint program you like for drawing the images that will make up a character animation. As long as you can save your images in IFF ILBM format, CITAS can use them. In addition, you don't have to worry about flipping and rotating images within the paint program because you can do this in CITAS.

When clipping a brush image from the paint screen (so you can save the image to disk), don't worry about cutting it to exact size. CITAS will automatically remove unnecessary blank space around the edges of the image. CITAS will also throw out any NULL (unused) bitplanes in the image, thus conserving memory.

Output Types

CITAS can produce 2 types of anim object output:

- Source file output in 'C' or assembler. This file includes complete information about the object (data, control structures, etc.). You can then compile or assemble the source file so that you can link the resultant object file with your program. During program execution, you reference the object by the structure names generated in the source file.
- Disk-based anim file. This is the most flexible form of output. There are 3 forms of this output as well:
 - normal save output that can be reedited by CITAS. These files use a more efficient RLE compression than standard IFF ILBM files since they can compress across horizontal lines of data.
 - final output that cannot be edited by CITAS (for distribution). Final output files are encrypted as well as compressed.
 - locked final output files that cannot be edited by CITAS (also for distribution). Locked files can only be loaded by the anim loader included with your copy of the GameSmith Development System; guaranteed to keep those anim files away from prying eyes.

There are three distinct advantages to using disk-based anim objects versus linked objects.

- Objects can be loaded/unloaded at any time. This allows your program to maintain in memory only those objects that are needed at one time. It also allows your program to make use of a HUGE number of "levels" or areas in a game, because all necessary objects can be loaded before a game level, and freed at the end of the level.

- Your program has dynamic control over how many elements are to be allocated in the object array at load time.
- Anim objects can be modified without regard to program code. Graphics and programming can remain completely separate.

CITAS Objects & the Animation System

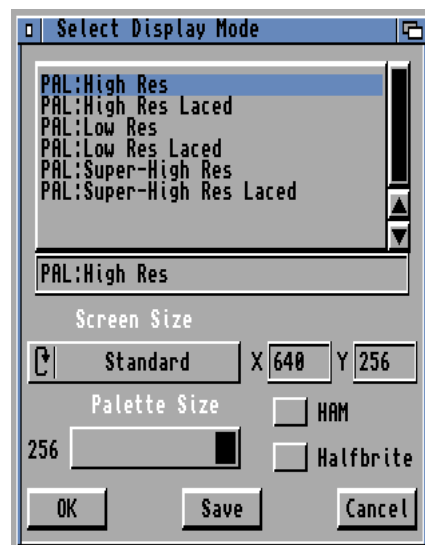
Objects created with CITAS can be used directly by the GameSmith animation system without any setup or processing by your program. This includes both linked objects and disk-based objects. You need only add an object to a display list by passing the address of its controlling structure to the animation system. This is true object oriented handling of animation objects, and lends itself to a number of unique applications. For instance, it's possible for you to completely change the disk-based anim objects used by a program without ever changing the actual program!

Your Serial Number

You can view your product serial number by selecting "About" from the Project menu. Make sure you write this number in the space provided in this User's Guide manual, as you'll need it to obtain technical support.

Display Modes

You can run CITAS in any display mode supported by your system. Workbench 1.3 users are restricted to an old-style type mode requestor limited to old chip set display modes. Users of Workbench 2.0 and above are presented with the display requestor shown below:

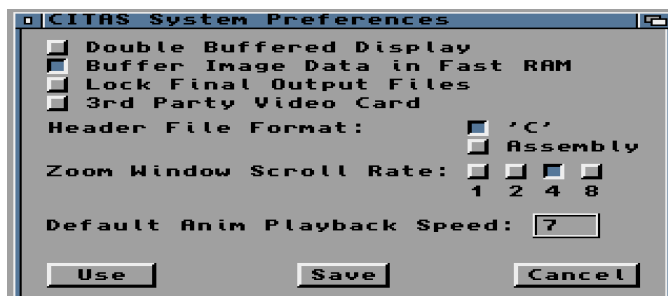


If the special HAM or Halfbrite modes are available in the selected display type, their respective buttons are enabled, and you may press them to enable those special modes. Note that Halfbrite **MUST** use a 32 color palette (6 bitplanes). HAM mode, on an AGA machine, can be either the old mode or the new AGA mode (6 or 8 bitplane usage respectively). Leave the palette slider at 64 to use the new HAM8 AGA mode (only available to AGA users), or move it to 16 to select the old HAM mode.

If you have not previously saved a display setting, the current Workbench display mode will be selected when the display requestor appears. You may save a display mode preference so that the requestor always comes up with the desired mode as the default. This is a handy time saving feature, and is very useful when working on that large game project.

System Preferences

CITAS allows you to set up and save certain preferred operating parameters. To display the preferences window, select "Preferences" from the Project menu.



Double Buffered Display

CITAS can use a double buffered display. This display method allows completely smooth animation preview, but requires 2 screen bitmaps (twice the Chip RAM). If your preferences setting is for a double buffered display, and there is only enough Chip RAM for a single buffered display, CITAS will fall back to a single buffered display and inform you of this action.

Buffer Image Data in Fast RAM

You may opt to load image data to Fast RAM instead of Chip RAM. With this setting, you are limited only by the amount of Fast RAM in your system in how many anims you may have loaded at one time. This will generally be the preferred method, as image data does not consume valuable Chip RAM space. However, buffering images to Fast RAM requires that the CPU copy images to the screen and not the Amiga's blitter Chip.

On some systems like the 3000, this will be faster anyway, but on stock old chip set Amigas, this display method may be too slow for previewing animations with large images. In this case you will want to load image data to Chip RAM.

NOTE: CITAS looks at this setting when loading image data, previously saved anims objects, and complex anim objects. If you change this setting in the middle of building an anim, animation preview within CITAS may be awkward and nonuniform. This setting has nothing to do with how your objects will display in your own program (see Anim Options).

Lock Final Output Files

Any final output files saved while this option is active (button depressed) will contain a lock. Locked anim files are specific to your unique serial number, and can only be loaded by the version of the anim loader provided with your system (see the Library Reference and the lock_loader module). This means that only programs YOU compile and link can use the results of your hard work. This is the highest security level available for anim objects.

Header File Format

When CITAS saves an anim object, it also saves an include file with names and labels that your program may optionally take advantage of. Set this in accordance to your language environment.

Zoom Window Scroll Rate

The "zoom" window is the magnified version of the image displayed when setting collision detection areas and points. The scroll rate is the number of pixels (large version) that CITAS will shift the image when the mouse pointer is placed against the zoom window borders. You will want to play with this to determine which works best on your system. Stock Amigas with a 68000 CPU may want to set this to the maximum amount allowed: 8 pixels.

Default Anim Playback Speed

While previewing an animation, you may change the playback speed by hitting the up arrow to play faster, and the down arrow to play slower. You can view the current value or type in a new one in the preferences window. This value is in vertical blank delays between image display updates.

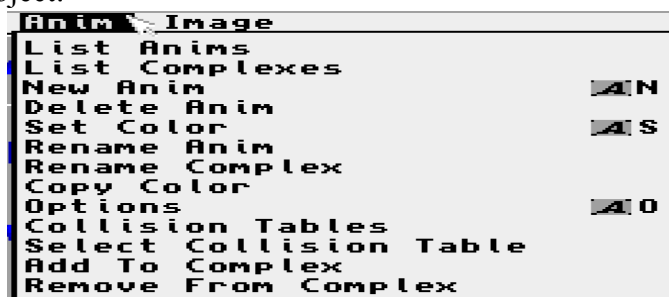
Creating a Simple Anim Object

A "simple" anim object, as defined by the GameSmith Development System, is an object composed of one single animation sequence (any number of image cells). Flipping through the anim cell frames in rapid succession creates the illusion of movement, or animation. This is traditional, or Disney style animation typically found in most video games, and is the basis for all anim objects currently available to GameSmith users. This section deals with the issues involved in creating such an object for use with the GameSmith animation system.

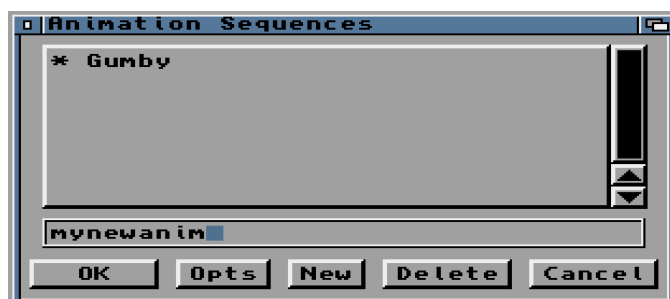
Creating a New Anim Object

There are 3 ways to start building a new anim object:

- When no anims are loaded in memory, simply load an image. CITAS will prompt you for the name of the new anim.
- Select "New Anim" from the Anim menu. Again, CITAS will prompt you for a name.
- Select "List Anims" from the Anim menu, type in a unique name in the input box, and press the New button.



Note that anim names **MUST** be unique. CITAS cannot know about anim names for objects still on disk, but it will enforce unique names for anims loaded in memory. This is because if you intend to take advantage of the header files produced by CITAS, thereby referencing objects by name, you should make an effort to make sure that every anim object in your game has a unique name.



You may notice that some anims in the list window are shown with an asterisk to the left of the name. This indicates that the anim has been modified since it was last saved. If you try to exit CITAS before saving any changes, CITAS will display a warning message.

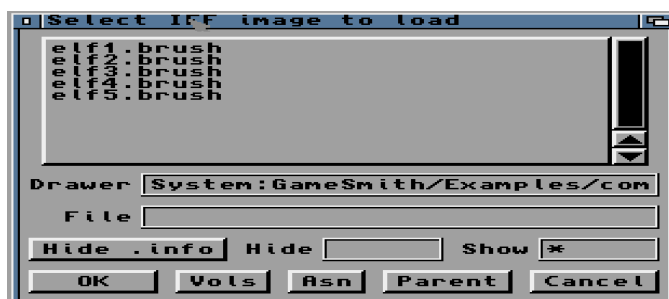
If you included any spaces in the anim name, CITAS will replace them with an underscore ("_") character. This is because symbol definitions within programs cannot include spaces, and neither can file names (the anim name is used as the file name and is included in the output header file). You should also make sure you don't use any special characters in the anim name that might prevent you from saving the object to disk or referencing it by name within a program.

Once the new anim is created, it becomes the current, or active anim. All operations take place on the current anim (add images, image manipulation, collision detection, etc.). The name of the current anim is shown at the top of the screen in the title bar.

Loading and Sequencing Image Cells

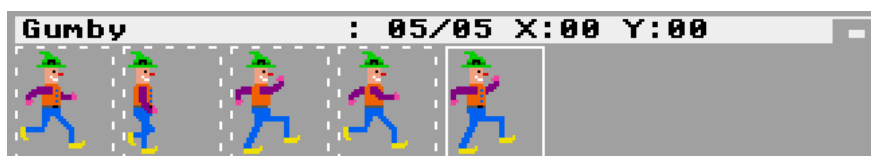
A single image within an animation sequence is often referred to as a "frame", or a "cell". Once you have created a new anim (see previous section), you're ready to load the frames that will make up the animation. Select "Load Image" from the Project menu, and a file requestor will appear, asking you to select an IFF ILBM image to load. CITAS only recognizes the IFF ILBM file format.

This example lists 5 images of an elf running right. For this reason, we'll name our new anim "elf_right". We'll want to load all of the images in order to create a smooth animation. CITAS will add each new frame to the end of the animation sequence. You'll notice that the last loaded is framed in a solid rectangle. You'll also notice that any other cells in the anim are framed with a dotted line rectangle. just as there is always one active anim within CITAS, so is there always one active cell within the anim (if any anims or cells exist in memory).



As you load the images that make up the animation, the cell size may change (the rectangle might get bigger) to accommodate a larger image. The smallest cell size is just large enough to accommodate the largest image in the anim.

Images are tiled across the screen from left to right; then down row by row. If there isn't enough room on the screen to hold all of the images in your anim, then they simply flow onto the next "page". You may easily move from one cell to the next by using the left & right arrow keys, or by simply positioning the mouse pointer over the cell you want and clicking the left mouse button. The current cell number, total cells in the anim, and the current cells X & Y offsets (more on that later) are shown in the title bar area of the screen. Note that in order to move to the next page of cells (if any exist), you will need to use the right arrow key. Likewise, in order to move to the previous page of cells, you will need to use the left arrow key.



Repositioning Image Cells

What if you didn't load the cells in the proper order? No problem. If you need to reposition some of the cells, there are two menu options to handle this: MOVE and SWAP. Select "Move", and CITAS will replace the title bar with the message "Move cell n before which image?", where 'n' is the current cell number. Use the mouse pointer to click on the new image position, or use the arrow keys to move to the new position and hit enter. The Esc key will cancel the operation. Likewise, you may swap two cells with the "Swap" option. If you swap cell 5 with cell 2, cell 5 becomes the new cell 2, and cell 2 becomes the new cell 5. The Esc key will also cancel this operation.

Copying Cells

In our elf example, only those images that were absolutely necessary were drawn in DPaint. Cell 2 needs to be copied and then moved before cell 5, leaving us with 6 cells of smooth animation. Select the "Copy" option from the Image menu. Note that duplicate cells have the distinct advantage of not requiring any additional memory for image data when used in a program. CITAS, however, does need to make an actual copy of the image data, but keep in mind that this will not be the case when the anim is actually used in your program. You should make use of duplicate cells whenever possible.

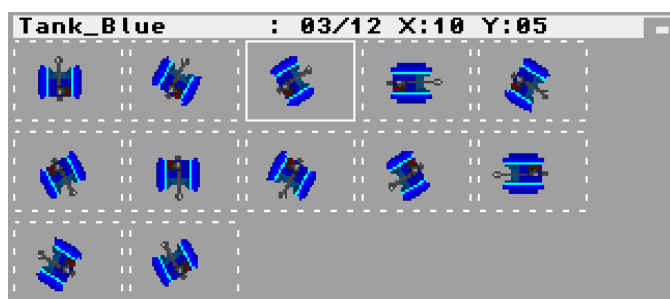
Image Orientation

You may flip or rotate images from within CITAS. A good example of this feature would be to create a whole new anim from our "elf_right" by simply flipping all images horizontally. The result is an elf running left. The entire process of creating our "elf_left" anim would be:

- Save the elf_right anim to disk.
- Rename elf_right to elf_left.
- Flip all cells horizontally.
- Check all cells for alignment (more on that later).
- Save elf_left to disk. That's it!



You've seen what flip horizontal will do. Flip vertical will turn the image upside down. The rotate option will rotate the image 90 degrees.



IMPORTANT! Any Flip All or Rotate All operation will keep duplicate cells intact, but flipping or rotating a single image that either was duplicated or is a duplicate will negate any duplicity for that cell.

Cell Info

To view more information about a particular image cell, select the "Info" option from the Image menu. You will see actual image dimensions, what type of memory the image data occupies (Fast or Chip), and whether or not the image is a duplicate.

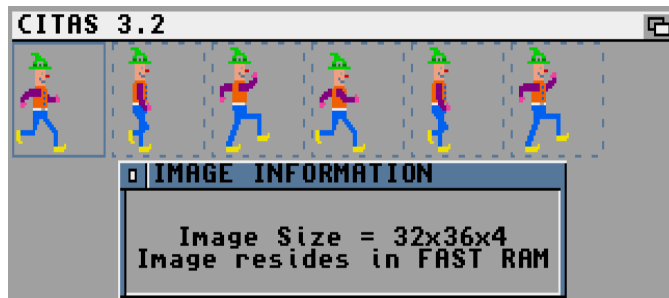


Image Colors and Anim Colors

Generally you will draw all of your images using the same color palette. Likewise, all images that will appear on a given screen should use the same color palette, or some will not appear as they should (wrong colors). We cannot stress strongly enough on the importance of careful color palette design BEFORE creating your game images.

When you load a new image from disk, the current anim takes on the colors of that image. If you don't want to use those colors, activate the cell whose colors you do want to use and select the "Set Color" option from the Anim menu. This will set the anim's color palette to that of the current cell.

CAUTION! When you save an anim object to disk, only the current color palette is saved. A color palette is not saved for every image in the anim. Therefore it is crucial that you set the color palette correctly before saving your anim object.

You may also copy one anim's color palette to another. For this operation, select the "Copy Color" option from the Anim menu. CITAS will present a list of target anims for the copy process. Select one or click on Cancel. The selected anim will then become the active anim so that you can see the effect of the new color palette on the images in that anim. If it looks awful, don't worry, you can bring back the original colors by simply selecting the "Set Color" option from the Anim menu.

It is important to remember that when you save the anim object to disk, you're defining the color palette for the object. If you load it again later, you won't be able to bring back any previous color palette (although even that can be solved easily enough by simply loading an image with the correct colors and setting the color palette for the anim).

Image Alignment

After all of the anim images are loaded, you will more than likely need to align them so that they play back smoothly. Images may be shifted in their cells by holding down the shift key and pressing the arrow keys. The X & Y cell offsets for the current image are shown in the title bar. The best way to get images properly lined up is to arrange them roughly in what looks like good positioning, and then fine tune their placement during animation preview (more on that later).

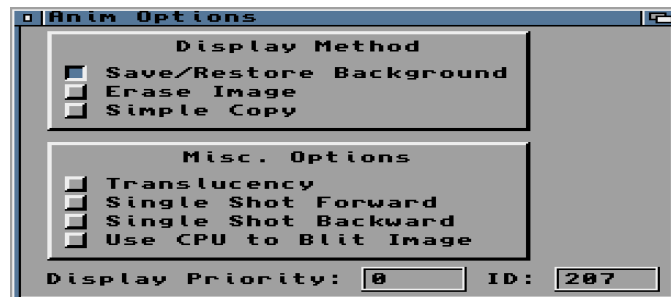
Cell Size

As stated previously, the smallest cell size will be large enough to accommodate the largest image in the anim sequence. Additionally, image widths are rounded up to the nearest word length (multiple of 16). This has to do with how the blitter chip accesses memory. One word (16 bits) is the smallest amount the blitter can deal with. In other words, you won't be able to shift the largest image unless you first adjust the cell size. You can do this simply by holding down the Ctrl key (control key) and pressing an arrow key. You may expand and contract cell sizes in multiples of 8 pixels.

Note that expanding a cell does not affect the actual image size. No additional memory is required for larger cell sizes. A larger cell size may be required so that you can align images properly. You may notice that by expanding the cell size not all images can fit in the same display. In this case, you are left with more "pages" of cells for the anim. This also does not require any additional memory, but is merely a display restriction.

Anim Options

You may set all other non-visible properties of an anim through the "Options" selection from the Anim menu.



Save/Restore Background

This is the default display method, and allows objects to move over a background scene without disturbing it (see the discussion of save areas in the User's Guide). Each time an object moves or animates:

- the background data in the previous location is restored
- the background data behind the object at the new location is saved
- the object image is copied to the bitmap.

This is the slowest method of the three because it involves 3 steps with every movement. However, it is the only one that allows background data to remain undisturbed during object movement.

Erase Image

This method does not save and restore background data. Instead, with every movement or animation of an object:

- the image is erased from the bitmap using the blitter mask (the area where the image was located is cleared to zero, or background color),
- the image is copied (using the blitter mask) to the new location.

This method is really only effective in a display without background data, or when background data does not occupy the same bitplanes as the image data.

Simple Copy

This is the fastest, simplest method. Unlike the other two display methods, the blitter mask is not used, and the entire rectangle of image data is merely copied to the destination bitmap. Moving an object of this type in a bitmap will leave a trail, or copy, of itself as it moves.

Translucency

A translucent object allows light to pass through it to one degree or another. For instance, a window, colored or otherwise, can be seen through, and objects behind the window are visible. A translucency effect can be achieved by merging image data with background data (or overlapping objects). Normally, when an object is copied to a bitmap, the bitplanes behind an image which are unaffected are cleared to zero. This guarantees that the image will appear with the proper colors. For instance, if an object is composed of 2 bitplanes (4 colors), using planes 0 and 1, and is copied to a bitmap with a depth of 4 bitplanes, the image area in bitplanes 2 and 3 are cleared to zero when the image is copied to the bitmap. When translucency is turned on, however, this is not the case, and bitplanes 2 and 3 would remain unchanged during the copy operation. A properly designed color palette can yield fascinating results in translucency mode. This mode sets the ANIM_MERGE bit in the anim structure, which can also be set or cleared via function calls.

Single Shot Forward

Normally, when animating forward (default), anim objects are allowed to "wrap" so that after the last cell is displayed the animation starts over again with the first cell. Sometimes this is undesirable. If you want animation to stop with the last cell, set this option. When animating the object within your program, the object will animate through the last cell and stop until you manually reset the sequence back to cell 0 (or whatever). This can be a very useful automatic feature.

Single Shot Backward

This option works exactly like the single shot forward mode, but applies only to objects animating in reverse. Normally, when animating an object in reverse, the sequence wraps from the first cell to the last, and continues to loop the animation forever. If you want animation to stop with the first cell, set this option.

Use CPU to Blit Image

By default, objects will be copied to a bitmap by the Amiga's blitter chip. You may opt to have the CPU handle copying instead of the blitter chip by setting this mode. Objects that are copied using the CPU have the advantage of being loaded to Fast RAM (if available), but usually take longer to display. It's worth noting that the AGA blitter chip will outperform even a 25MHz 68040 at image display.

Display Priority

When your program adds an object to a display list, it is inserted into the list according to its priority setting. The priority defines the order in which the animation system will draw objects into the bitmap, thus allowing some objects to pass in front of or behind other objects.

The higher the priority, the higher the object is stacked on the screen (objects with a higher priority pass over the top of objects with a lower priority). Zero is the lowest possible priority. Objects with like priorities are simply drawn in the order in which they were added to the list, with the last object appearing in front of the previous, and so on. The anim priority field is a 16 bit unsigned value, which means object priorities may range from 0 to 65535 (a 64K range).

Anim ID

When you create a new anim or rename an existing anim, a new ID value is assigned to that anim. The current ID value is kept in the CITAS.config file. The ID field is an unsigned 16 bit value, and thus may contain a value from 0 - 65535 (64K range).

CITAS automatically increments the current ID value for new anims, so you should never have to modify this field. Only if your program were to use more than 64K objects would this be a problem. CITAS objects should always have a unique ID number because the anim name is defined to be the ID value in the header file.

Renaming an Anim Object

You may rename an anim object at any time through the "Rename Anim" option in the Anim menu. Note that doing so causes CITAS to increment the ID value for the object. This is because typically, a rename is used to create a copy of an existing anim before images are reoriented (flipped or rotated). The new ID value guarantees your objects will have unique ID's.

Selecting Another Anim

Memory permitting, CITAS can handle any number of anims simultaneously. If additional anims are loaded before this one, the minus sign ("-") will appear in the title bar. Hitting the minus key will bring you to the previous anim object. If additional anims are loaded after this one, the plus sign ("+") will appear in the title bar. Hitting the plus key will bring you to the next anim object. A faster way to reach any particular anim object is to display a list of all anims and select the one you want. You can display this list through the "List Anims" option of the Anim menu.

Animation Preview

CITAS wouldn't be very useful if you couldn't preview animation. You can animate the active object at any time from any cell. Obviously, an anim object needs more than one cell in order to animate.

Object Animation

To preview an object's animation, simply hit the up arrow key. The object will start animating forward, starting with the current cell, at the current speed setting. To animate the object in reverse, hit the "<" key. To switch back to forward animation, hit the ">" key. While animating, the up arrow will speed up playback, and the down arrow will slow it down. The current setting will be reflected in the System Preferences window. To end animation preview, hit the Enter key or click the left mouse button.

Note that single shot animations act accordingly in preview mode. That means that if you animate an object forward that has the single shot forward option set, animation will stop with the last cell. The same applies to single shot backward mode. When aligning single shot anims, it's sometimes best to line up all images first, and then set the single shot modes.

Pausing and Fine Tuning Image Alignment

While previewing an animation, hitting the space bar will pause the animation at the cell currently displayed. This status will be indicated in the title bar, along with the current cell number and its X & Y offsets. At this time, you may shift the current image in its cell by holding down the shift key and using the arrow keys.

To single step through the animation, let up on the shift key and use either the left or the right arrows. The left arrow key moves backward through the animation, and the right arrow moves forward through the animation. In this way you can check correct image placement by quickly flipping between two images, making adjustments as needed until they are lined up to provide smooth animation.

To show the animation at normal speeds again, simply hit the up arrow key. While paused, you can cancel animation completely by hitting the return key or by clicking the left mouse button.

Creating a Complex Anim Object

There is another type of anim object available to GameSmith programmers called a complex anim, or anim complex (often shortened as cplx). The anim complex is a logical grouping of simple anims, and may contain any number of simple anims. For detailed information on this object type, please refer to the User's Guide. It might also help to glance at the structure hierarchy chart of the animation system in the User's Guide as well.

The anim complex is comprised of one or more simple objects, although only one simple anim, or anim sequence, is active at any given time. This allows easy and logical control over very complex animated objects in your program. As an example, let's take a Street Fighter type character: to the player, a fighter is just that, ONE fighter, but with many different animated moves. In actuality, the fighter is composed of many animated sequences, from punches, to kicks, to walking or jumping.

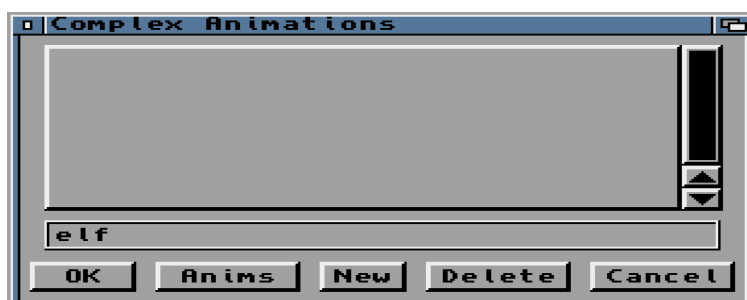
Using CITAS, you can combine all of these animated sequences, each defined by a simple anim, so that to your program it is also a SINGLE controllable anim object. Your program simply tells the animation system which anim sequence to display and animate at any given time, with reference to the object's anim complex structure.

Creating a New Anim Complex

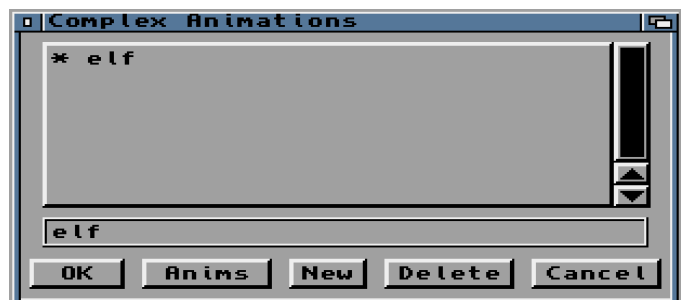
After you've created all of the anim sequences (simple anims) that will make up your anim complex, you will need to tell CITAS that you want to create a new anim complex. Select "List Complexes" from the Anim menu. A list window will appear showing all of the anim complexes in memory. You can create a new one simply by typing a name in the input box and clicking the New button.

Just as anim names must be unique, so must the anim complex name be unique. It must not be the same as any other anim or anim complex name currently in memory.

Again, those objects in the list that have been modified since they were last saved will appear with an asterisk next to the name.

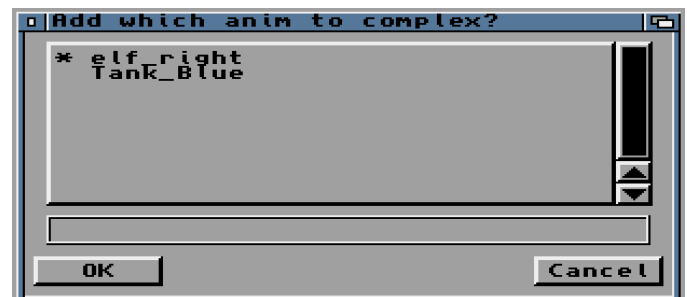


Now leave the name in the input box and click on the Anims button. This will display a list containing the names of all of the anim objects that reside within the anim complex. Since this is a new anim complex object, it won't contain any anims just yet (the list is empty). Click on the Add button to bring up a list of all anim objects loaded in memory that are not already assigned to an anim complex.



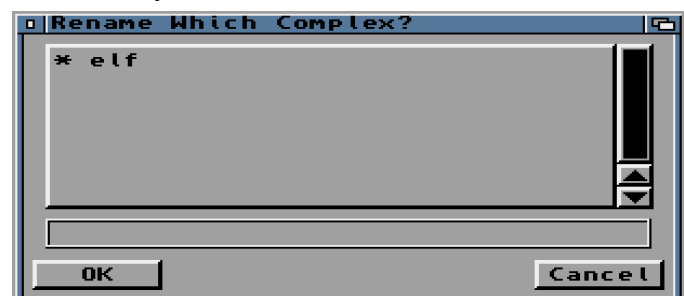
Repeatedly add anims to the anim complex until the complex contains all of the anim sequences that your object requires. You may add or remove anims at any time.

That's it! You've just created an anim complex. You may now save the complex and/or generate source code so your program can use it.



You will generally always want to save your work as a disk based file. The anim complex, once saved to disk, contains all of the individual anim sequences as well as one color palette and one collision table (if any). You should know that the color palette and collision table saved come from the first object that was added to the complex. Since the anim list is alphabetized, this will not necessarily be the anim at the top of the list. When building a complex anim, all simple anims should use not only the same color palette, but the same collision table (if any) in order to avoid any confusion.

Note that you can also rename an anim complex at any time by selecting the "Rename Complex" option from the Anim menu.



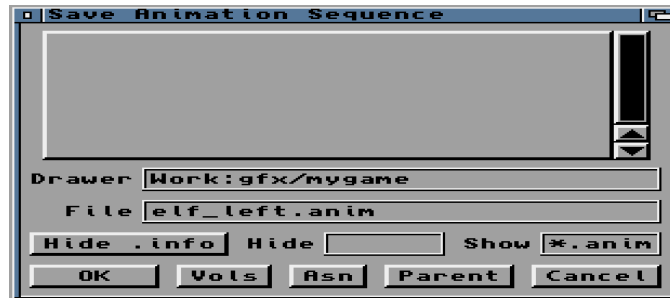
Complex anim objects have a unique ID associated with them just like simple anims. However, you cannot edit or change the ID field of an anim complex. CITAS will assign a unique number to the ID field of the anim complex structure in the same manner as it assigns numbers to simple anims. Like simple anims, the anim complex name is associated with the unique ID number in the header file written to disk .



As a final note, you should make sure that every anim within the complex uses the same display method, and probably the same processor as its display engine (CPU or blitter chip).

Saving Your Work

When building your anims or anim complexes, you may save your work at any time, so that you can quit CITAS and pick up where you left off at a later date. You may save anims, anim complexes, and collision tables separately, although any anims or anim complexes which make use of a collision table save the table in the anim file as well.



Anim Files

When you select "Save Anim" from the Project menu, CITAS will prompt you for a name under which to save the CURRENT anim. The current anim is the one shown on the screen. The default is the name of the anim, plus a ".anim" extension. In addition, CITAS will also save another file with the name of the anim plus a ".h" or ".i" extension, depending on which header output format you've selected in System Preferences. A ".h" file will be produced for 'C' header files, and a ".i" file will be produced for assembler include files. See the section below for information on header file contents.

The following components are saved in the anim file:

- all image data
- the color palette currently shown on the screen
- the entire collision table used by the anim (if any), including collision type names
- all collision detection areas (if any)
- all collision detection points (if any)

Image data is compressed using a simple run length encoding algorithm; the same method used in IFF files, although anim files typically achieve greater compression rates.

The anim file contains all of the information that the `gs_load_anim()` function needs to build an entire anim object for use by your program. Everything is self-contained, so your program need not bother with any of the details of the object itself.

Final Output

When you select "Save Anim Final" from the Project menu, the results are the same as Save Anim, but with the following differences:

- The file cannot be edited again by CITAS. This is a security measure for object distribution purposes. Other owners of the GameSmith Development System will not be able to edit your objects for their own use.
- The file is encrypted. This is another security measure, hiding the actual image data in the file.

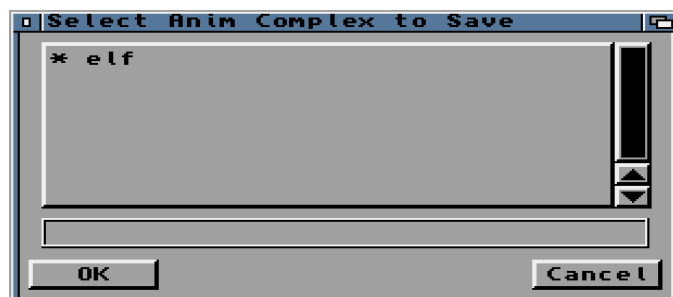
Final output files, by default, are saved with just the anim name without an extension (example: `elf_left` instead of `elf_left.anim`). Your program may load these objects in the same manner as non-final output anim files. The `gs_load_anim()` function will load both impartially.

Locked Final Output Files

In the System Preferences window, there is a button which allows you to lock final output files. When this button is ON (pressed down), any anims saved as final output will be locked to your unique serial number. The files are identical except that you will need to link your program with the included lock_loader module in order to load them. This is the highest level of security.

Anim Complex Files

When you select "Save Complex" from the Project menu, CITAS will display a list of all anim complexes in memory, and allow you to select the one you want to save. The default file name is the name of the complex plus a ".cplx" extension. In addition, CITAS will also save a header file with the name of the complex plus a ".h" or ".i" extension, just like when saving a simple anim.



When a complex anim is saved, the file is built in the same manner as when saving a simple anim, except that all of the animation sequences that make up the complex anim are saved, not just a single anim.

Final output anim complex files and locked final output file operate in exactly the same manner as simple anims.

Your program may load complex anims or simple anims with indifference. You need only check the type field of the anim_load structure to determine the object type (simple anim or complex anim).

Anim Header Files

As mentioned, when an anim file is created, CITAS also saves a header file for the anim in either 'C' or assembly format. These header files contain name definitions that your program may use (optional). Use of logical names within your program conveys meaning, thus making your programs more readable and maintainable. However, you should note that the use of logical names for object ID fields does not lend itself to object interchangeability (you have to recompile your program if the object completely changes).

Where simple anim objects are concerned, these header files contain a name equated to the ID field of the anim, any names for collision areas, and any names for collision points. Complex anim object headers contain all of the same information plus names for the individual anim sequences within the complex. These are extremely handy.

A simple anim header with no collision detection might look like this:

```
#define ELF_LEFT_ANIM      203 /* label/ID of anim */
```

or in assembly:

```
ELF_LEFT_ANIM EQU        203 ;label/ID of anim
```

That's it, but the name can be very useful. Merely include the file in your program to make use of the name. The name of the anim in this case was simply `elf_left`. CITAS causes all letters to be capitalized and appends the `"_ANIM"` to the end of the name. This denotes an anim ID.

The file might look something like the following when using a collision table, and when naming collision areas and points:

```
#ifndef CTBL_ELFGAME
#define CTBL_ELFGAME          1
#define ELFGAME_ELFHEAD      0x00000001
#define ELFGAME_ELFTORSO     0x00000002
#define ELFGAME_ELFARM       0x00000004
#define ELFGAME_ELFLEGS      0x00000008
#define ELFGAME_BOW          0x00000010
#define ELFGAME_ARROW        0x00000020
#define ELFGAME_ACID         0x00000040
#define ELFGAME_LAVA         0x00000080
#define ELFGAME_GOLD         0x00000100
#define ELFGAME_SWORD        0x00000200
#define ELFGAME_BADHEAD      0x00000400
#define ELFGAME_BADTORSO     0x00000800
#define ELFGAME_BADARM       0x00001000
#define ELFGAME_BADLEG       0x00002000
#endif
#define ELF_LEFT_ANIM 203 /* label/ID of anim */
#define COBJ_HEAD1          0
#define COBJ_RIGHTARM1      1
#define COBJ_LEFTARM1       2
#define COBJ_BODY1          3
#define COBJ_LEGS1          4
#define CBGD_FORWARD_FOOT1   0
#define CBGD_BACKWARD_FOOT1  1
```

You can see that the entire collision table is defined (the table only contains 14 of the possible 32 type definitions). Since the headers use conditional compilation/assembly (`#ifndef`), you can include any number of anim headers safely even though they define the same table. Also defined in this example are a few collision detection areas, a few collision detection points, and the anim ID. The object-to-object collision area names are preceded with `"COBJ_"`, and the object-to-background collision point names are preceded with `"CBGD_"`. The numbers associated with each correlate to the area field of both the `collision_struct` and the `coll_bg_struct`.

Anim complex header files contain all of the same information, but with the following additions: the name of every anim sequence contained in the complex is also defined. For example:

```
#define ELF_CPLX            204 /* label/ID of anim complex */
#define ELF_LEFT           0   /* sequence # of anim within complex */
#define ELF_RIGHT          1   /* sequence # of anim within complex */
#define ELF_LEFT_ANIM 203    /* label/ID of anim */
#define ELF_RIGHT_ANIM 202    /* label/ID of anim */
```


Collision Tables

If you want to be able to detect collisions between objects in animation display list, you'll need to set up a collision table. Within the collision table, you may define up to 32 different collision area "types". A collision area is merely a rectangle placed on or around one cell in an anim sequence. Every cell may contain any number of collision areas. Every collision area must be one of the available 32 types. The collision types allow you to enable collisions only between certain areas, such as monster leaches colliding only with the hero's leg, and not with each other. They also allow your program to more readily determine what has collided with what.

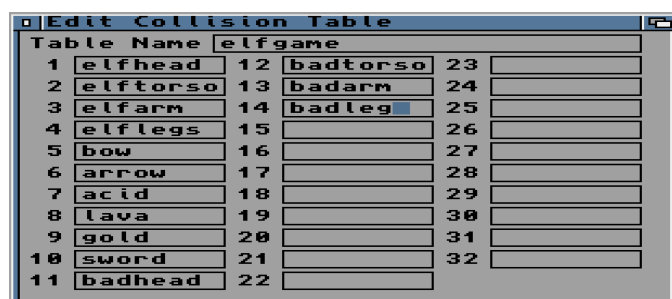
Setting up collision detection should be the LAST step in creating the anim objects for your program. At the very least you should not attach your collision table to any of your objects until they are all finalized. It is very important that you design you collision tables carefully, as once they are attached to all of your objects it can be cumbersome to make changes.

Building a Collision Table

To build a new collision table, select "Collision Tables" from the Anim menu. This will display a list of all collision tables in memory (normally you'll only need one). Decide on a unique name for your collision table (like anim objects, collision table names must be unique), type it into the input box, and click on the New button.



You've now created an empty new collision table. Now you must define the collision types within your new table. Leave the table name in the input box, and click on the Edit button. You are presented with the table name and 32 input boxes for type names. You may move freely between the input boxes and table name box with the tab key or by clicking on the boxes with the left mouse button. You may change the table name here as well. Note that although only 8 characters are shown in each box at a time, the input boxes actually allow much longer names, and can be scrolled left and right.

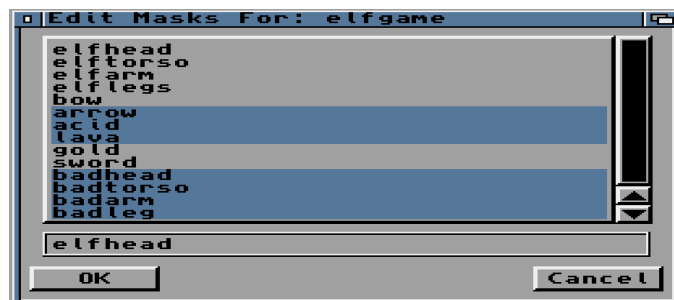


This example depicts a simple game where all of the possible object-to-object collision types are defined in only 14 slots. You need not make use of all 32 types.

Once you've finished defining your table, click on the close window gadget in the upper left hand corner to return to the collision table list window.

Collision Masks

Now you will need to define which types can actually collide with each other. This is called the collision mask. By default, none of the types can collide with each other, meaning no object-to-object collision events will happen. Click on the Masks button. For each type in the table, you will be prompted to select those other types with which it can collide.



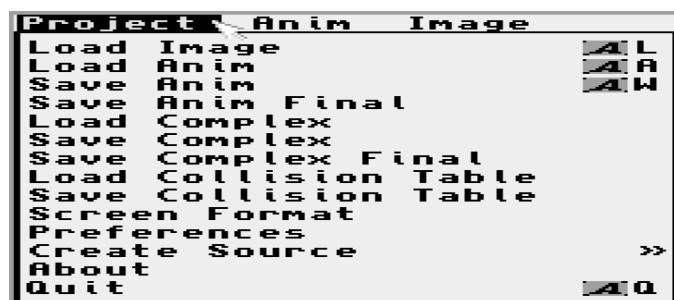
The selected collision types that are enabled will be highlighted in the list window. Once you are finished with one type, move on to the next by clicking on the OK button.

After you've finished defining the last area type collision mask you will be taken back to the collision table list window.

Saving

You may save your collision table to disk independently of an anim object. Simply select "Save Collision Table" from the Project menu for a list of tables in memory.

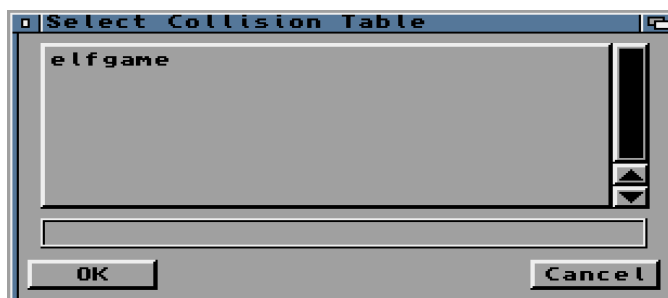
By default, the collision table will be written to disk as the table name plus a ".ctbl" extension.



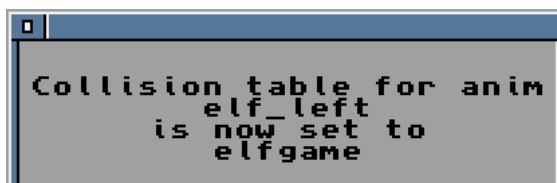
Selecting For Anim Use

After you've finished the collision table, you'll next need to "attach" it to those anims that will use it. You cannot define object-to-object collision areas on an anim until you have attached a collision table to the anim sequence.

Choose "Select Collision Table" from the Anim menu. You will be presented with a list of collision tables currently loaded in memory. Select the appropriate table.



Once you've selected a table for your anim, CITAS will inform you that the current anim is attached to the chosen collision table.



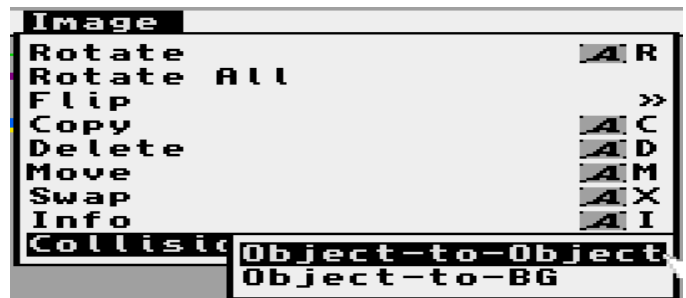
Repeat this process for all of your anim objects.

Note that once a collision table is attached to an anim object, the table is saved in the anim file. You will not need to save the table independently.

If an object on disk and an object in memory both use the same collision table, and you modify the table in memory before loading the 2nd object, CITAS will detect the difference in the collision tables, and prompt for a new name for the 2nd table. This just emphasizes the importance of careful collision table design.

Object-to-Object Collision

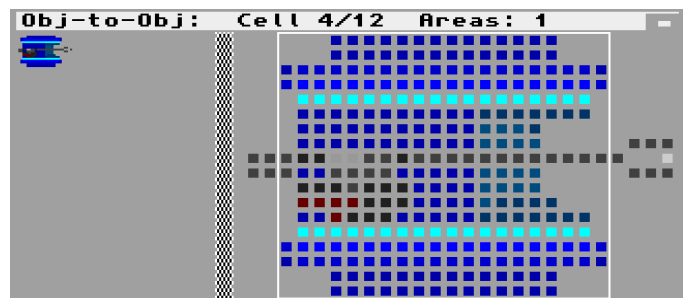
As stated in the last chapter, object-to-object collision detection areas are merely invisible rectangles placed over your images. You may place any number of these rectangles on each image of your anim objects, and each area may use any of the possible 32 different types. Then, when 2 collision enabled area types overlap, the animation system will notify your program of this event by calling your collision detection function. It's really all very simple. For instance, in a sword fighting game, if a sword area connects with a neck or head, well, it's probably time to play that "off with his head" animation sequence.



The Zoom Window

To enter object-to-object collision area definition mode, select "Object-to-Object" off of the "Collision Areas" selection of the Image menu. Note that CITAS will only let you enter this area if you have previously selected a collision table for the current anim.

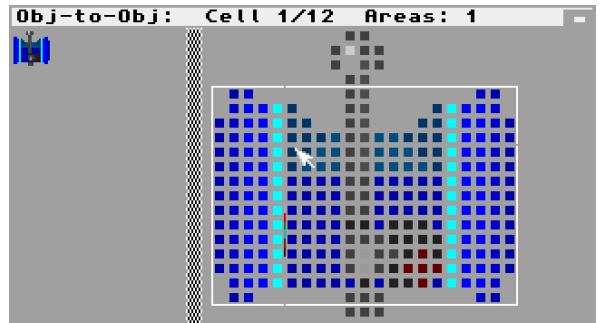
The collision area/point definition screen is divided vertically into two sections. The large section on the right shows a magnified view of the current image, and is known as the zoom window. The smaller section on the left shows the image in actual size (as much as will fit in the left window). Placed over the actual sized image you will see a small square. This shows you what portion of the image is being displayed in the zoom window.



Scrolling

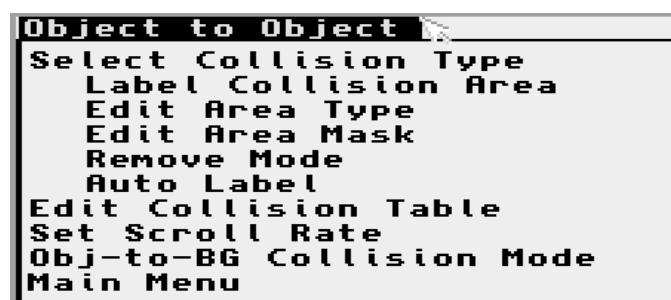
You may move around in the zoom window simply by moving the mouse pointer against the edges of the zoom window. This will scroll the image. Note that to scroll left, you will need to place the mouse pointer over the vertical bar on the screen which defines the left edge of the zoom window. If you feel that the scroll is too slow or too fast on your machine, you may want to adjust the pixel speed at which the image scrolls. To do this, choose "Select Scroll Rate" from the menu. You have a choice of 1, 2, 4, or 8 pixels per scroll. This is reflected in the System Preferences window as well.

You may also position the zoom window over a specific place on the image by clicking on the actual sized image and holding down the left mouse button. Now the box over the image will move with your pointer. Let up on the mouse button to refresh the zoom window. If the image is too large for all of it to be displayed in the left window, you may scroll the image by moving the box to the left or right edges of the window. You may always scroll the zoom window a single pixel at a time by holding down the shift key and using the arrow keys.

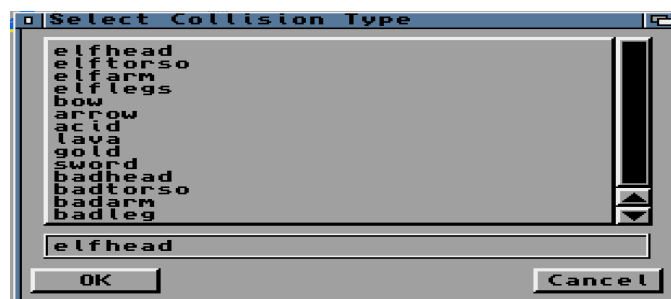


Selecting a Collision Area Type

Before you can define a collision detection rectangle, you must first choose its type. When you first entered the object-to-object collision detection screen, you were asked to select an area type.



You may also select a new type at any time by using the "Select Collision Type" menu option. Now any areas you define over the image will be of the selected type.

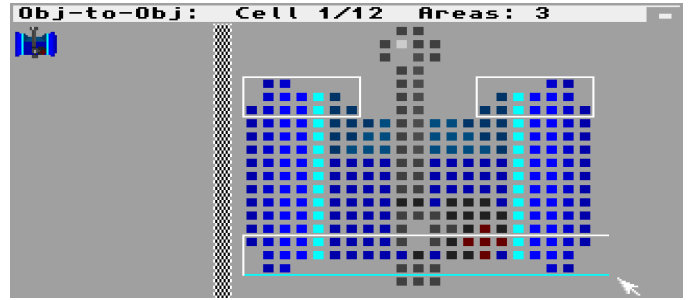


Defining a Collision Area Rectangle

You'll notice that when you move the mouse pointer over the zoom window, a set of cross hairs appears showing you exactly which pixel your pointer is over.

To start defining a collision rectangle, hold down the left mouse button. Now you can drag and resize the box over your image. You can also scroll around on the image while resizing your collision rectangles. When satisfied, let up on the left mouse button.

The title bar will reflect the additional collision detection rectangle that was just added. You will not be able to start a new rectangle inside of another, but you are allowed to overlap them. If you need to delete some areas, simply select "Remove Mode" from the menu. A check mark will appear next to the menu selection. Now you may delete collision detection areas by simply clicking on them in the zoom window.



Labeling a Collision Area

There are two ways to label collision detection areas. The first, and easiest, is to turn on "Auto Label" mode. You turn on this mode by selecting the menu option of the same name. While this mode is on, a check mark will appear in the menu next to the Auto Label option. Now every time you define a collision detection rectangle, you will be prompted for a name to be associated with the area.

You may also label existing rectangles by selecting "Label Collision Area" from the menu. When in this mode, when you click on an existing rectangle you will be prompted for a new label for that area. You can modify area names at any time by using this method.

Modifying an Existing Area

Besides being able to modify the label for an area, you may also change the area type and collision mask. To modify area types, select "Edit Area Type" from the menu. Then whenever you click on an existing area, you will be prompted to select a new type for that area out of the types in the collision table. To modify area collision enable masks, select "Edit Area Mask" from the menu. Then whenever you click on an existing area, you will be prompted to define the collision enable mask for that area. Note that this does NOT affect other areas of the same type. This capability allows you to set up special case collision enable masks on your objects. When you first define a collision area, the default collision enable mask for that type is used.

Selecting a New Image

You don't have to go back out to the main menu in order to select another image in the anim sequence. You can move between all of them simply by using the left and right arrow keys.

Object-to-Background Collision

Sometimes it's handy to be able to determine whether an object is colliding with "background" data, or data that is part of the static display (not other anim objects). For instance, if you want to determine if your character is standing on grass or water, this method may be used. Object-to-Background collision detection involves defining invisible points, or pixels on or near the image where you want to detect background data. Background data may be collision enabled on a per bitplane basis.

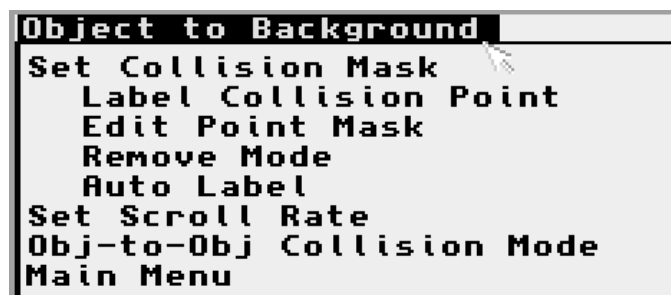
The display screen used to define object to background collision points is the same as it is for object-to-object areas. Therefore, this section will not reiterate the display or how to scroll around within it.

Setting the Collision Enable Mask

When you first enter the object-background collision screen, you will be prompted to select the collision enable mask for the points you will define. You are presented with a window containing 8 buttons. Each button, when pressed (as shown), enables collision detection with its corresponding bitplane of the bitmap. The default is collision with all bitplanes enabled(as shown). This window encompasses bitmaps up to and including 8 bitplanes in depth. If the actual target bitmap is not as deep as the mask you have defined, that's OK. The mask is simply applied to those bitplanes of the bitmap in which you are displaying your object.



You may change the active collision enable mask at any time by selecting the "Set Collision Mask" menu option.



Defining Collision Points

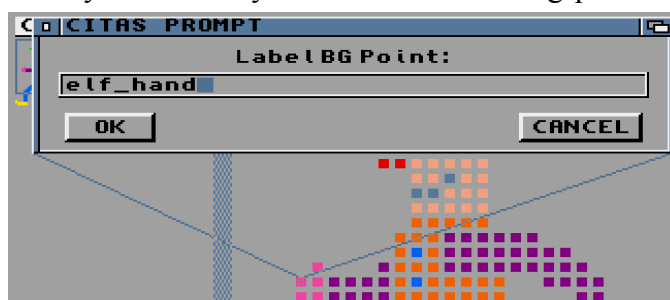
You will notice when you move your mouse pointer into the zoom window that there appears a little flashing square under the pointer. It surrounds one single magnified pixel. If you click the left mouse button, the box will stay behind, surrounding the pixel, but will also contain an 'X' through it. This indicates that a collision point resides on that pixel. The title bar reflects how many collision points are defined for the image.

Like object-to-object collision detection, you will need to place collision points on every single image that you want to collide with background data.

To delete a collision point select "Remove Mode" from the menu. A check mark will appear next to the menu option. Now whenever you click the mouse button over an existing collision point, it will be deleted. You may hold the mouse button down, and move around over the image, erasing all collision points underneath the pointer. To turn off "Remove Mode", click on the menu option a second time. The check mark will disappear.

Labeling a Collision Point

Just like you can label collision areas, you may label collision points. Again, there are two ways to accomplish this. If you select "Auto Label" from the menu, whenever you define a new collision point you will be prompted for a name for the point. You may also modify the names of existing points by selecting "Label Collision Point" from the menu. While in this mode, clicking on a collision point will cause the label prompt to appear, showing the existing label (if any).



You'll notice that CITAS draws lines from the edges of the prompt window (bottom or top, depending on point location) to the point which you are labeling. This is so you know exactly which point is being affected.

Modifying an Existing Point

If you want to change the collision enable mask of an existing point, or merely want to display its current setting, select "Edit Point Mask" from the menu. While in this mode, clicking on an existing collision point will cause the collision enable mask prompt to be displayed, allowing you to modify it for that point. Note that this does NOT affect the active collision enable mask setting.

Source Output

If you prefer, you may link your anim objects directly into your program instead of loading them from a disk file. To accomplish this, you will need to generate source code for your objects, which can then be compiled or assembled, and linked with your program. CITAS is capable of generating source code in either 'C' or assembler format. The source output contains all of the same information kept in an equivalent disk file.

Advantages of linked anim objects:

- Objects are automatically loaded at run time. No additional code is required to load them.
- The program can be distributed as one file.

Disadvantages of linked anim objects:

- Objects cannot be freed from memory.
- Object array size cannot be dynamically controlled at run time.
- There is no possibility of object interchangeability.

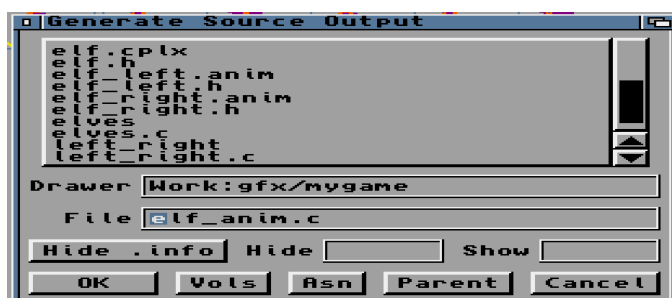
Generating Anim Source

Once you've finished your anim object and saved it to disk, select "Create Source" from the Project menu, and choose the correct language format. If the current anim sequence is part of an anim complex, CITAS will notify you of this and ask whether you want to generate source for the entire complex. If you select NO, then a source file will be created with just the current anim object in it.



Next, CITAS will prompt you for the number of objects it should build in the output array. Anim objects can always be referred to as being an array. This can be an extremely useful and memory saving feature, as multiple objects of the same type can be generated that all use the same image data. It is also time saving in that CITAS will build every object in the array in one simple step. For instance, generating a simple anim object will produce an array of `anim_struct` structures. Generating a complex anim object will produce an array of `anim_cplx` structures.

By default, the output file name will be the name of the object plus "_anim" and a language extension. The language extension will be ".c" for 'C' source files, and ".asm" for assembler source files.



Source Code Labels

CITAS chooses various labels for each type of structure or array produced in the source file. These differ slightly depending on language use. To reference them, you need only declare them as external symbols within your program. Once the object is linked with your program, you may reference these symbols like any other variable.

This reference key will help you interpret the source code symbol protocols described in the following pages:

{obj_name}	The name of the output object. This will be the anim name if the object is a simple anim, or the complex name if the object is an anim complex.
{anim_name}	The name of the anim sequence (simple anim).
{cell_num}	The cell number within an anim sequence, from 1 to n.
{area_num}	The collision area/point number, from 1 to n.
{cplx_name}	The name of the anim complex. This will only be included in the file when generating source for an entire anim complex.
{array_count}	The number of elements in the output array.

NOTE: all label names are forced to lower case letters for source output.

‘C’ Symbols

```
color palette (red,green,blue values respectively for each palette entry):
    unsigned char {obj_name}_color[]
```

```
image data* (one cell/frame):
    unsigned short __chip {anim_name}{cell_num}_image[]
    unsigned short __chip {anim_name}{cell_num}_mask[]
```

```
coll_bg_struct (obj-to-background collision point):
    {anim_name}{cell_num}_cbgd{area_num}
```

```
collision_struct (obj-to-obj collision area):
    {anim_name}{cell_num}_cobj{area_num}
```

```
blit_struct:
    {anim_name}{cell_num}_blit
```

```
anim_struct:
    {anim_name}_anim[{array_count}]
```

```
anim_cplx:
    {cplx_name}_cplx[{array_count}]
```

*image data arrays will not contain the "__chip" keyword if the CPU will be used to blit the object (See Anim Options).

Assembler Symbols

color palette (red,green,blue values respectively for each palette entry):

```
SECTION data,DATA
{obj_name}_color
```

image data* (one cell/frame):

```
SECTION chip,DATA_C
{anim_name}{cell_num}_image
{anim_name}{cell_num}_mask
```

coll_bg_struct (obj-to-background collision point):

```
SECTION data,DATA
{anim_name}{cell_num}_cbgd{area_num}
```

collision_struct (obj-to-obj collision area):

```
SECTION data,DATA
{anim_name}{cell_num}_cobj{area_num}
```

blit_struct:

```
SECTION data,DATA
{anim_name}{cell_num}_blit
```

anim_struct:

```
SECTION data,DATA
{anim_name}_anim
```

anim_cplx:

```
SECTION data,DATA
{cplx_name}_cplx
```

*image data will be placed in the DATA section, and not the DATA_C section if the CPU will be used to blit the object (See Anim Options). The DATA_C section specifies Chip RAM.

Additional Header Files

Source generation produces an additional header file beyond that normally created when saving the object to a disk file. The file name will be the name of the object plus "_anim", followed by a language extension. The extension will be ".h" for 'C' source output, and ".i" for assembler source output. The files vary depending on language and contain:

'C' Files

```
#define {OBJ_NAME}_COUNT      (array_count)
#define {OBJ_NAME}_COLORS      {palette_entries}
```

Assembler Files

```
{OBJ_NAME}_COUNT EQU {array_count}
{OBJ_NAME}_COLORS EQU {palette_entries}
```

These macro definitions/equates are forced to upper case letters.

Keyboard Controls

During Normal Display:

+	(including keypad) move to next anim sequence in memory.
-	(including keypad) move to previous anim sequence in memory.
Right arrow	move to next image in the sequence (you may also click on the image with the mouse).
Left arrow	move to previous image in the sequence.
Up arrow	begin animation playback/preview.
Shift right arrow	shift image one pixel to the right.
Shift left arrow	shift image one pixel to the left.
Ctrl right arrow	expand image cell size horizontally.
Ctrl left arrow	contract image cell size horizontally.
Ctrl down arrow	expand image cell size vertically.
Ctrl up arrow	contract image cell size vertically.

During Animation Playback:

Up arrow	increase playback speed.
Down arrow	decrease playback speed.
Enter	(including keypad) stop playback (you may also click the left mouse button).
Space	pause playback.

During Playback Pause:

Right arrow	display next cell in sequence.
Left arrow	display previous cell in sequence.
Up arrow	continue normal (automatic) playback.
Shift arrows	same as while not animating (shift image one pixel).
Enter	end playback.

While Setting Collision Areas/Points:

Right arrow	display next cell in sequence.
Left arrow	display previous cell in sequence.
Shift arrows	scroll zoom window by one magnified pixel

- Hitting the enter key while a requestor window is displayed implies hitting the OK or YES button.

7 The Sound System

Introduction

The GameSmith sound system is a very sophisticated set of routines designed to handle all aspects of Amiga sound except music soundtracks. The fundamentals of soundtrack capabilities are already in place, and may be supported in future versions of the system. Given the easy availability and popularity of existing soundtrack editors and players such as MED, Soundtracker, Noisetracker, Protracker, etc., it is fairly easy to use those existing tools for playing music, and the GameSmith system for game sound effects.

The GameSmith sound system runs in the background, transparent to your program, needing very little attention other than to start sounds playing. The GameSmith sound system contains the following features:

- Loading of IFF 8SVX sound samples, including stereo samples, and samples with loop data
- Loading of raw sound samples (non-IFF)
- Interrupt driven automatic playback of samples in the background
- Playback of sound samples from Fast RAM
- Playback of sound samples in any octave
- Unlimited sound sample size
- Automatic handling of looping samples for real-time echo
- Automatic special effects such as fades and frequency changes
- Total dynamic user control of samples in progress

Operation

The GameSmith sound system has several distinct pieces: User callable routines, the Vertical Blank Interrupt Controller, and the Software Interrupt RAM Cache.

User Callable Routines

This is your interface to the sound system. When you open the system, the interrupt drivers are installed, which allows automatic playback of sound samples in the background. Closing the sound system removes the interrupt drivers. These routines allow you to load sound samples from disk, start and stop sounds, dynamically modify sound channel values, and retrieve sound channel status information.

Vertical Blank Interrupt Controller

This is the heart of the GameSmith sound system. This code controls playback of sound samples through the Amiga's sound hardware, as well as dynamic changes such as fades, loops, and frequency ramps (see the `sound_struct` for details). Changes can be made, and samples started and stopped, at a precision of once every vertical blank.

Vertical blanks take place 60 times a second on NTSC machines, and 50 times a second on PAL machines. This controller also dictates caching events to the software interrupt if your program makes use of this feature. On systems without Fast RAM, there is no advantage to using the caching feature.

Software Interrupt RAM Cache

The GameSmith sound system has the ability to play back sound samples from Fast RAM. If you intend to load sound samples to Fast RAM (if available), then you should initialize the sound system with a Chip RAM cache. This is a very powerful feature which uses only a small portion of precious Chip RAM for sound, freeing up more space for graphics data. This software interrupt system handles the copying of sound data from Fast RAM to the Chip RAM caches as needed.

For those unfamiliar with the Amiga's audio hardware, this is a necessary step, since the hardware can only reference sound data in Chip RAM. The vertical blank controller monitors the usage of the caches, and causes the software interrupt system to execute when one or more caches need to be filled. Since sound data plays relatively slowly when compared to other operations within the Amiga computer, this caching scheme is very efficient, with very low overhead. You will not notice any performance degradation when using the Chip RAM caches as opposed to sample playback directly from Chip RAM.

The Sound Structure

The GameSmith sound system requires a sound structure associated with every sound sample. The sound structure contains information necessary for playing the sample:

```
struct sound_struct
{
    unsigned char *data;
    unsigned char *loop;
    unsigned long length;
    unsigned short repeat;
    unsigned short count;
    unsigned short period;
    unsigned short volume;
    unsigned short type;
    unsigned short flags;
    short volfade;
    unsigned short volcnt;
    short perfade;
    unsigned short percnt;
};
```

The following describes all of the fields within the sound structure and how they are used. Note that the load routines initialize all fields except flags, so any special effects setup must be filled by your program after the sample has been loaded.

data

This is a pointer to the actual sound data. It may reside in either Fast RAM or Chip RAM, depending on how you load the sample. This pointer must contain an even address (the sound DMA hardware deals in 16 bit words).

loop

This is a pointer to the loop portion of the sample, and must also be an even address. If the sample being loaded does not contain a loop portion, then this pointer will be the same as the data pointer. If you play a sample with a loop portion more than once, the effect is an echo. Beginning with the second playback of the sample, the data is restarted at the loop point. Your program can modify this value after loading a sample to produce realtime echo effects with absolutely no overhead.

length

This is the length of the sound data in 16 bit words (unsigned shorts). Although the Amiga's sound DMA hardware has a limit to the length of data that can be played through a sound channel, the GameSmith sound system removes this restriction by updating the channel's data pointer when required. Sound data is limited only by the amount of available contiguous RAM.

repeat

You set this field to the number of times you want the sample to play. A value of 0 will play infinitely. The sound sample loaders initialize this field to 1.

count

This is a running total of the number of times the sample has been played. While the sample is playing, your program can monitor this field.

period

The sound period is an Amiga term which specifies the data output rate through the sound DMA channels. This determines the sound's frequency or pitch. IFF 8SVX files contain this information, but raw sound files do not. Approximate note value periods are defined in the files NOTES.H and NOTES.I. You can use the note periods to play a musical tune with any sound sample. The lower the number, the higher the pitch.

volume

This specifies at what volume the sound sample will play through a channel. Valid volume numbers range from 0 (silent) to 64 (loudest). IFF 8SVX files contain volume information, but raw sound files do not. The volume field of a raw sound sample is set to maximum (64) after loading.

type

At this time, the only valid type of sound accepted by the GameSmith sound system is type SND_EFX. This field is initialized by the loaders.

flags

At this time, the only flag available to the programmer is the SND_FAST bit. Setting this bit before loading a sound sample tells the loader to try and load the sample to Fast RAM. If not enough Fast RAM is available (or none exists), then the sample is loaded to Chip RAM.

volfade

If this value is nonzero, then the volume of the sound sample is decremented by VOLFADE amount after every VOLCNT interval. Negative values cause a volume increase rather than a decrease.

volcnt

If this value is nonzero, then the sound volume will be decremented by VOLFADE amount after every VOLCNT number of vertical blank interrupts. If this value is zero, then the sound volume will be decremented by VOLFADE amount after every complete pass of the sound data.

perfade

If this value is nonzero, then the sound sample period is decremented by PERFADE amount after every PERCNT interval. Negative values cause a period increase rather than a decrease.

percnt

If this value is nonzero, then the sound period will be decremented by PERFADE amount after every PERCNT number of vertical blank interrupts. If this value is zero, then the sound period will be decremented by PERFADE amount after every complete pass of the sound data.

Sound Functions

<u>Function</u>	<u>Description</u>
gs_sound_open	Open the interrupt driven sound system
gs_sound_close	Close the sound system
gs_start_sound	Play a sound sample through a channel
gs_stop_sound	Stop any sounds on a channel
gs_set_volume	Change the volume of a sound playing on a channel
gs_sound_period	Change the period of a sound playing on a channel
gs_sound_check	Check the status of the sound channels
gs_load_raw_sound	Load a file from disk as raw sound data
gs_load_iff_sound	Load an IFF 8SVX sound file from disk
gs_free_sound	Release sound sample data

8 Utility Functions

In addition to specific systems, the GameSmith Development System includes a number of very handy utility functions.

The AmigaDOS Librarian

The GameSmith AmigaDOS Librarian provides a simple, clean method of working with AmigaDOS ROM libraries. The librarian automatically keeps track of libraries which are opened, and closes all of them with one easy function call. In addition, program modules that need to make sure a given library is opened need only call the open function to make sure that the library is opened.

The librarian first checks to see if the library had been previously opened, and if so, skips the open request. Thus a program may safely open a library any number of times; though only the first call actually opens the library. Before exiting, one call to the close function will close all libraries that were opened by the GameSmith librarian.

The Librarian and ‘C’

Most ‘C’ compiler startup code will automatically open libraries which it thinks your program will need. This is a very convenient feature, but only as long as you’re using only the supplied functions that come with the compiler. When linking with other routines outside of the compiler, such as those contained in the GameSmith linker library, the compiler cannot tell which libraries to open automatically, and so you must specifically open the libraries yourself. In any case, it’s always best to handle AmigaDOS libraries yourself to guarantee the correct libraries are opened, and to guarantee compatibility with previous operating system levels.

Simply include the header file LIBPTRS.H from the GAMESMITH:INCLUDE/LIBRARIES directory, and use the GameSmith librarian to open the necessary libraries. Including this file should prevent the compiler from defining its own AmigaDOS library base pointers and opening those libraries. For a list of libraries supported by the GameSmith librarian, see the file LIBRARIES.H in the same directory.

The Librarian and Assembler

Normally, handling AmigaDOS libraries from assembler is tedious at best. With the GameSmith librarian, you no longer need to define the library names, and allocate space for the library base offsets. Simply include the file LIBPTRS.I from the GAMESMITH:INCLUDE/LIBRARIES directory, and call the librarian to open and close libraries. In addition to ease of use, this will make your code much cleaner; especially if you’re opening more than one library. For a list of libraries supported by the GameSmith librarian, see the file LIBRARIES.I in the same directory.

CYDEC Encryption Module

A large concern for commercial software production shops and amateurs alike is the distribution of graphics and sound files in a form that the end user can rip off. Often times commercial art will find its way into hacker demos, distributed on public bulletin boards, or whatnot. The same holds true for sound files. The standard Amiga IFF graphic and sound files are wonderful to work with because they are universal. Paint programs and sound programs alike output in the same format - the CORRECT format for Amiga computers. The best paint and sound sample software all output files in these formats. The GameSmith system can easily read them so that you can use them. But this wonderful universal format

becomes a problem when you want to distribute the proprietary work you've labored so long to produce. How do you make your graphics and sound files "safe" from prying eyes? Simple: encrypt them!

Operation

CYDEC stands for cypher-decypher. It is this module that allows you to encrypt IFF graphics and sound files. The concept of encryption is to change the data format so that it is no longer recognizable by other programs or people. The two included programs `gs_cypher` and `gs_decypher` do just what their names imply. Each automatically recognize IFF graphic and sound formats, encrypting or decrypting each accordingly. The decryption program is provided in case you want to transform a file back to its original state. Note that each will replace the original file.

The functions `gs_cypher`, `gs_decypher`, the GameSmith IFF picture loader, and the GameSmith IFF sound loader all call the CYDEC module. In addition, the source code for CYDEC is included and is user programmable.

That means you can change the encryption/decryption algorithms to anything you want, making the format all your own, and keeping those files private. Only your programs will be able to load them. You need only recompile it after a change, and then link with `gs_cypher.obj` and `gs_decypher.obj`.

You can then use `gs_cypher` to encrypt your files. Just link your `cydec.o` module with your other program modules and the picture and sound loaders will automatically and transparently recognize and load your encrypted files. Here's how it works:

File Types

IFF ILBM = 0

IFF 8SVX = 1

Encryption Process

- `gs_cypher` calls `gs_cypher_block` (a function in CYDEC) with the file type to get an encryption header. Make sure this is unique, as the header is used to identify your encryption method. The header may be any length up to 4096 bytes. This call is the key to resetting and selecting the encryption algorithm. Subsequent calls to the encryption function must assume that the proper encryption algorithm is the one selected with the last call to `gs_cypher_block`.
- While rewriting the data file, `gs_cypher` calls `gs_encrypt_data` to process the data before writing it back out.

Decryption Process

- The `loadILBM`, `load_iff_sound`, or `gs_decypher` functions call `gs_identify_cypher` with the encryption header found in the file and its length. If your program recognizes the header, then subsequent calls to the decryption function must assume the correct decryption algorithm is the one last selected from `gs_identify_cypher`. Again, when `gs_identify_cypher` is called, this is your key to reset the algorithm.
- If the encryption header was properly identified, then `gs_decrypt_data` is called to process the file data as it is read. The files `gs_cypher.obj` and `gs_decypher.obj` have been prelinked to include everything except the functions in `cydec.o` and some in the GameSmith.lib linker library.

The following are examples of how to relink them after changing the CYDEC source code and recompiling:

```
slink gs_cypher.obj cydec.o lib GameSmith:GameSmith.lib
slink gs_decypher.obj cydec.o lib
      GameSmith:GameSmith.lib
```

Refer to the library reference for information on correctly linking with the CYDEC module. If your programs will not be loading encrypted files, you need not link with the CYDEC module, as "stub" routines are provided in the GameSmith linker library.

Note that although `gs_cypher` and `gs_decypher` replace the original file, they use a temporary file while processing. Your files will not be corrupted should an error occur.

Encryption Hints

Even if you don't want to change the algorithm, change the header string in the CYDEC source code and recompile. This will make your encrypted files your own. Nobody else will be able to load them. Remember to relink with `gs_cypher.obj` and `gs_decypher.obj` before encrypting your files!

If you change the encryption algorithm intended for sound files, test out the encrypted version using the SND program supplied on the example disks (without relinking it to recognize your encryption method). It should sound nothing like the original file. The supplied encryption method does this properly.

WARNING! NEVER change the `cydec_proto.o` file. The CYDEC functions will always be called in the same manner by `loadILBM`, `load_iff_sound`, `gs_cypher`, and `gs_decypher`

Joystick Polling

You may easily poll the Amiga's two joystick ports at any time, with a single function call `gs_joystick`. The polling routines support both single button digital joysticks and multiple button digital joysticks. The current possible joystick return values are as follows. Note that the values are bit assignments, and more than one may be set at a time (for instance a combination of up and left would equal a diagonal upward and left direction).

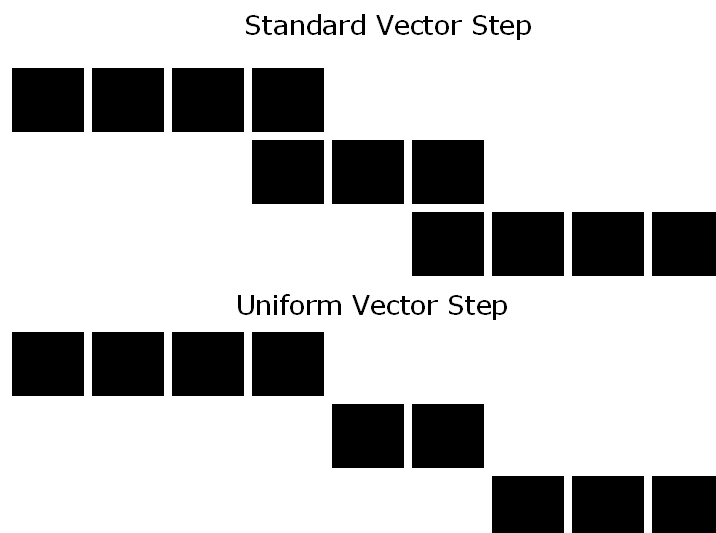
JOY_RIGHT	Joystick pushed to the right
JOY_LEFT	Joystick pushed to the left
JOY_DOWN	Joystick pulled down (towards the user)
JOY_UP	Joystick pushed up (away from the user)
JOY_BUTTON1	The first button is depressed. Please try to cheer it up.
JOY_BUTTON2	The second button is depressed. Please try to cheer it up. Write a great game and cheer them both up!

Vector Routines

The GameSmith vector routines allow a number of simultaneous, controlled precision 2D vectors. A vector is merely a line that has direction. The GameSmith vectors allow controlled precision of a line which moves from an origin X,Y coordinate to a destination X,Y coordinate. Up to 100 vectors may be in simultaneous operation.

Once the vector has been initialized with a beginning point and end point, you control movement and rate of movement along the line. The vector step routines simply return the new coordinates along the line. Your program can make use of these coordinates in any way you like. Some good applications include plotting a line in a bitmap, and moving an object in a straight line at any angle.

Two types of vector step routines are provided: a standard line step, and a more uniform line step. The standard step should be used when plotting visible lines, as it will appear better on the screen. The uniform step should be used for everything else, because it provides more uniform movement along the vector.



Random Number Generator

The random number generator will return a 16 bit short integer pseudo random number in the range of 0 to n, where n is the input parameter. An input parameter of 0 reseeds the random number generator from the system clock, but is not necessary as the function is self-seeding. As computer generated random numbers go, the included function is very good.

Task Priority

A function is provided for setting your programs task priority. Valid AmigaDOS task priorities range from -128 (lowest) to 127 (highest).

Audio Filter Control

Two routines are provided for turning the low pass audio filter on and off. When the low pass audio filter is on, the system power LED is bright. When the filter is off, the power LED is dim.

File Requestor

A file requestor is provided in the GameSmith.lib that you may use in your programs. It is the same one used by CITAS, and is compatible with all operating system levels from 1.3 on up. However, this file requestor can only be linked with the SAS/C compiler link library because it makes use of certain handy pattern matching functions provided by the 'C' compiler.

Vertical Blank Timer

A very simple vertical blank timer is provided. Once installed, it merely increments a counter with each vertical blank interrupt. You can poll this value, and you can reset the counter to zero.

Vertical Blank Functions

Included in the package are easy calls for adding any function to the system's vertical blank server chain. You may specify a server priority from -128 to 127 as defined by the Amiga's Exec. After the function is added, it will be executed during every vertical blank interrupt period, which occurs 60 times a second on NTSC machines, and 50 times a second on PAL machines. You must remember to remove your functions from the server chain before exiting.

Plot Routines

Fast plot pixel routines are included to replace the sloth like ReadPixel and WritePixel functions provided by Commodore Amiga. A function is also provided to invert a pixel (flip all color bits). These are very fast routines.

Bitmap Allocation

Two simple routines are provided for allocating and freeing a bitmap and its controlling structure. Use of these routines will guarantee that the resultant bitmap is properly aligned for any machine, therefore making them extremely useful. Old style bitmap allocation will not work correctly for all AGA display modes. These routines correct the problem by calling the new 3.0 bitmap allocation function in the appropriate situation.

Utility Functions

<u>Function</u>	<u>Description</u>
gs_open_libs	Open specified AmigaDOS ROM libraries
gs_close_libs	Close all opened AmigaDOS libraries
gs_random	Quickly generate a 16 bit random number
gs_joystick	Poll a joystick port
gs_task_prio	Set the programs task priority
gs_file_requester	Prompt the user for a path and file name
gs_LEDOn	Turn the audio filter on
gs_LEDOff	Turn the audio filter off
gs_version	Return GameSmith.lib revision number
gs_add_vb_server	Add a function to the vertical blank server chain at the given priority
gs_remove_vb_server	Remove a function from the vertical blank server chain
gs_open_vb_timer	Open and initialize the GameSmith vertical blank interrupt timer
gs_close_vb_timer	Close the vertical blank timer
gs_vb_time	Get the count value from the vertical blank timer Reset the timer clock count to zero
gs_init_vector	Initialize a vector for use
gs_step_vector	Get next vector point using the specified gradient
gs_ustep_vector	Get the next vector point using the specified gradient, but with more uniform movement

9 Using The GameSmith Library

The GameSmith Development System includes a pre-compiled linker library named GameSmith.lib. This library contains all of the GameSmith functions, and works equally well in both 'C' and/or assembler environments, as well as with any language that can use linker libraries and pass function arguments on the stack or in registers.

The library itself was created using the Object Module Librarian (OML) program from SAS Institute Incorporated. OML produces very efficient libraries, allowing the linkage process to quickly scan for necessary library modules. However, some older linker programs do not recognize this indexed library format. For this reason, another version of the linker library is provided. It is called GameSmith_ni.lib. If you experience problems linking with GameSmith.lib, try GameSmith_ni.lib.

Linking With The GameSmith Library

In order to use the GameSmith library functions in your programs, you must link with the GameSmith linker library. There are two ways to link with the GameSmith linker library. The first and easiest, if your programming environment will handle it, is to specify

```
GameSmith:GameSmith.lib
```

as a linker library that must be searched when creating your program. The other way is to create your programs in two phases; the compile phase (or assemble), and the linkage phase. Depending on the development system you are using, the name of the linker may vary.

Under the SAS/C development system, the linker is called SLINK. A typical link command line might look like this:

```
slink from lib:c.o myprogram.o lib lib: sc.lib,lib:  
amiga.lib, GameSmith:GameSmith.lib
```

This links with the normal SAS/C startup module, one or more of your own program modules, the SAS/C linker library, the Commodore Amiga linker library, and the GameSmith linker library to produce one executable program.

Under most other development systems for the Amiga, or versions of the SAS/C compiler prior to version 6 (when it was the Lattice compiler), the linker is called BLINK. All command line arguments for BLINK are exactly the same as they are for SLINK. A typical link command line for an assembler module might look like:

```
blink from myprogram.obj lib GameSmith:GameSmith.lib
```

If your link command will be too long to fit on one line, you can place all linkage information into a nicely formatted text file that the linker can interpret. The file would look something like:

```
FROM lib:c.o  
Mymodule1.o mymodule2.o mymodule3.o  
lib lib:sc.lib,lib:amiga.lib,GameSmith:GameSmith.lib  
TO myprogram
```

Pass the name of the link file to the linker like:

```
slink with myprogram.link
```

Note that the linker will only incorporate into your program those modules which it needs. For instance if you are referencing the animation system, but not the sound system in your program, only the animation module is included from the GameSmith linker library.

When working in a 'C' development environment, it is best to specify the GameSmith library as the last to be searched. This is because the GameSmith library contains replacements for some common 'C' functions which might be more efficient under other compiler environments. These replacements were necessary because some GameSmith library functions were written in 'C'.

Using Locked Anim Files

CITAS has the capability to produce locked object files for security purposes. These files are locked to your unique serial number, and can only be loaded and used in your program if you have the correct key. This unique key is contained in the file lock_loader.o, and must be linked into your program before you can load anim files that are locked to your serial number.

The following example illustrates this simple link process:

```
FROM
lib:c.o
mymodule1.o mymodule2.o mymodule3.o
lock_loader.o
lib lib: sc.lib, lib: amiga.lib,GameSmith:GameSmith. Lib
TO myprogram
```

Using Encrypted Graphics and Sound Files

Another "safety" feature built into the GameSmith system is the ability to encrypt IFF graphic and sound files. This is extremely useful when you want to keep those great graphic and sound files private - away from public scrutiny so as to avoid any modification. Again, in order to load these special files, you'll need to link with a separate object module. This one is called cydec.o. This file, unlike lock_loader.o is user modifiable, so as to allow any number of encryption techniques. Refer to the User's Guide for more information on this subject.

The following example illustrates how to link with cydec.o:

```
FROM
lib:c.o
mymodule1.o mymodule2.o mymodule3.o cydec.o
lib lib:sc.lib,lib:amiga.lib,GameSmith:GameSmith.lib
TO myprogram
```

Calling Library Functions From 'C'

There are a few considerations to be noted when calling GameSmith library functions from a 'C' program.

- All GameSmith structure definitions and function prototypes are included in the header file GameSmith.h. Additional function interfaces for passing input parameters in registers are prototyped in the file include/proto/all_regargs.h (see 3 below).
- Normally, 'C' programs under AmigaDOS have an integer size of 32 bits; the same size as a long integer. Your program modules that call GameSmith functions must use an integer size of 32 bits.
- If you are unsure of the integer size in your 'C' programming environment, write a quick program to print out the integer size using the following line:

```
printf("integer size = %d bits\n", sizeof(int)*8);
printf("long integer size = %d bits\n", sizeof(long)*8);
```

Both the integer and the long integer size should be 32 bits long. However, if the results of this program yield something other than 32 (like 16 or 8) for the integer size, then you need to include the following define at the top of your programs:

```
#define int long
```

This redefines the variable type int to actually be compiled as a long (long integer), which should be 32 bits. In addition, the size of a short must be 16 bits.

- All GameSmith functions accept parameters in ANSI standard format. That is, all parameters must be placed on the stack by the compiler before making the call to a GameSmith function. However, if your compiler has the ability to pass function parameters in registers, this allows slightly faster execution speed when calling a function. The additional header files in the include/proto subdirectory prefixed by "_regargs.h" are included for this purpose. Those GameSmith functions that were developed in assembler have an equivalent function prototype. These functions are prefixed with an underscore character ("_"). These files were set up for use with the SAS/C development system, and may require some modification for use with other 'C' compilers. Refer to your compilers documentation on passing function arguments in registers.
- GameSmith function parameters are ALWAYS 32 bits in length when called from 'C', regardless of how much of the 32 bits is actually used by the function being called. This is merely a GameSmith convention. For instance, a function that needs a 16 bit integer as its first parameter will still expect a 32 bit integer to be placed on the stack, even though only the lower 16 bits will be used. As is normal, all pointers are 32 bits in length.

Calling Library Functions From Assembler

There are a few considerations to be noted when calling GameSmith library functions from an assembler program.

- Most of the GameSmith library functions were written in assembler, and thus are easily called from assembler. However, some functions, where speed was not an utmost priority, were written in 'C'. These functions usually involve operations dealing with the disk drive or handy setup routines. However, because some functions were written in 'C' does not necessarily mean that they execute any slower than an equivalent function in assembler.

In order to call GameSmith functions that were written in 'C', you will need to place all input parameters on the stack before making the call. For instance, the following 'C' function would be called like this:

```
; call the function: int c_func(a,ptr,b);
; where a is an integer (32 bit long word),
; ptr is a pointer to something, and b is
; also a long integer (32 bit longword).
```

```
XREF _c_func
move.l d0,-(a7) ;push argument b
move.l a0,-(a7) ;push argument ptr
move.l d1,-(a7) ;push argument a
jsr _c_func ;call the function
add.l #12,a7 ;restore stack pointer
move.l d0,result ;save return result
```

- Note that the arguments are placed on the stack in reverse order, and that the stack needs to be "fixed" after the function returns in order to remove the arguments you placed there. Also, functions that were written in 'C' are always prefixed by an underscore character. This is invisible to 'C' programmers, but must be accounted for when calling the function from assembler.
- Functions that return a result always return that result in register d0.
- For safety's sake, always make sure that registers contain a full 32 bit value when calling GameSmith functions, even if the function will use only the lower 16 or 8 bits. This will avoid any possible sign extension errors or other misinterpretations which can be very difficult to debug.
- The include file GameSmith.i defines all structures used by the GameSmith Development System as well as all necessary equates and external function references. The file was written in a form usable by the Devpac Amiga assembler system. It should work with most assembler systems, but some modifications may be necessary; most likely in defining proper structure field offsets.

SAS 6.5 Users Note

SAS changed from 6.3 to 6.5, and the compiler now **ALWAYS** defines DOSBase. If you are using SAS C 6.5 or later than edit the `GameSmith:include/libraries/libptrs.h` file and comment out the line that defines DOSBase.

Understanding the Library Reference

The synopsis portion of the library reference describes the function name, return value, and necessary input parameters. If applicable, appropriate registers are indicated directly beneath the function line. If no registers are indicated, then the function was either written in 'C' or it has no return value and/or input parameters. Functions that were written in 'C' are usually indicated in the description of the function. Check the include files in the proto subdirectory (the files with a ".i" extension) for the name of the function if you are unsure (it may need a leading underscore character).

All example code in this book is in 'C' format. For those unfamiliar with 'C', the following is a quick rundown of some 'C' terminology, operators, and language syntax. This will help you to understand the library reference examples in this book.

char	A byte value (8 bit)
int	A 32 bit long word
long	A 32 bit long word
unsigned	Don't interpret the high order bit of the following char, int, or long as a sign (positive or negative)
*	The asterisk has many uses in 'C'. If used in the synopsis section in front of a variable name, it indicates a pointer to the variable type (struct, int, char, etc.). <code>struct anim_struct *anim; /* ptr to anim structure */</code> If used in the example section in front of a variable name (for example *count), it refers to the value pointed to by the pointer variable. If used between two values or variables, it can mean a multiply operation.
void	If appearing before a function name, it means that the function does not return a value. If appearing as input to the function (between the parenthesis), it means that the function has no input parameters.
struct	This refers to a structure. A structure is an object that contains other variables, and can even contain additional structures. In assembler, these fields would be referenced as offsets from the beginning of the structure. For instance, if register a0 contains the address of an anim_struct, you can reference the cell field like this:

```
move.w ANIM_CELL(a0),d0
```

In 'C', referencing the cell field from a pointer to an anim_struct would be accomplished this way:

```
temp = anim->cell;
```

If anim was declared as an actual instance of an anim_struct and not a pointer to one, the cell field can be referenced as follows:

```
temp = anim.cell;
```

#define In the examples used in this book, the 'C' macro definitions can be accomplished with an assembler equate. However, a 'C' macro definition can be used for much more than assigning a simple value to a name. The following lines accomplish the same thing:

```
#define SCREEN_WIDTH 320  
SCREEN_WIDTH EQU 320
```

;	End of a statement
{	Start of a code block. Used to begin a function, "if" segments, "for" loops, and "while" loops.
}	End of a code block.
/	Divide operation
/*	Begin comment
*/	End comment
=	Assignment. Prefixed with another operation, it can be used as a short form for a = a + b, for instance. The following statements are identical in operation: a*=b; and a=a*b;
==	Test equality
&	When appearing between two values, it means a logical bitwise AND operation. If prefixing a variable, it means the address of the variable.
&&	Boolean AND condition. Usually appears in an "if" statement
	Logical bitwise OR operation
	Boolean OR condition. Usually appears in an "if" statement
>	Greater than
<	Less than
>>	Bitwise shift right. For example a variable can be shifted right by 16 bits with the line: a>>=16; /* can also appear as "a = a>>16;" */
<<	Bitwise shift left
0x	This is the 'C' notation for representing a hex value. It precedes the actual hex number. For example, instead of the common Motorola assembler notation for the value \$5f, the 'C' representation would appear as 0x5f.

10 Library Quick Reference

The following lists all GameSmith library functions by category with a brief description of what the function accomplishes.

Graphics Functions

<u>Function</u>	<u>Description</u>
gs_loadILBM	Load an IFF ILBM picture file from disk
gs_get_ILBM_bm	Examine an IFF ILBM picture file and fill a bitmap structure accordingly
gs_chiprev	Inquire chipset version
gs_plot	Plot a point in a bitmap using the specified color
gs_plot_reverse	Exclusive OR a point in a bitmap
gs_plot_test	Get the color value of a point in a bitmap
gs_get_blitter	Allocate the blitter in a system friendly way
gs_free_blitter	Allow other tasks to use the blitter
gs_blit_image	Copy a graphic image to a bitmap using a mask
gs_blit_image_fine	Copy a graphic image to a bitmap using a mask and fine control variables
gs_blit_image2	Copy a graphic image to a bitmap using a mask after first calling gs_blit_save_bg
gs_blit_copy	Copy a graphic image to a bitmap without a mask
gs_blit_copy_fine	Copy a graphic image to a bitmap with fine control variables (no mask)
gs_blit_copy2	Copy a graphic image to a bitmap without a mask after first calling gs_blit_save_bg
gs_blit_save_bg	Save background data from a bitmap where an image will be placed
gs_blit_restore	Restore background data
gs_blit_clear	Remove an image from a bitmap by writing zeros over the image
gs_blit_get_save_area	Allocate a save area for background data
gs_blit_free_save_area	Release save area memory
gs_blit_memcpy	Copy a block of memory using the blitter
gs_blit_fill	Fill a Chip memory area with a value
gs_get_bitmap	Allocate a bitmap
gs_free_bitmap	Release a bitmap
gs_user_copper	Attach or change a user copper list in a display
gs_hard_copper	Build a hardware list out of a user copper list
gs_replace_ucop	Replace the current user copper list
gs_free_hard_copper	Free hard copper list

Display Functions

<u>Function</u>	<u>Description</u>
gs_display	Simply create a low level display
gs_create_display	Create a low level display with all parameters
gs_show_display	Show the low level display
gs_flip_display	Show the other view in a double buffered display
gs_old_display	Show the display that was showing before creating a low level display
gs_remove_display	Release and remove a low level display
gs_LoadRGB	Reload entire color palette with new values
gs_SetRGB	Reload a single color palette entry with a new value
gs_scroll_vp	Scroll a viewport by delta X and Y
gs_scroll_vp_pf1	Scroll playfield 1 of a viewport

<code>gs_scroll_vp_pf2</code>	Scroll playfield 2 of a viewport
<code>gs_scan_copper</code>	Incorporate any changes to display copper list
<code>gs_alloc_svpv</code>	Allocate memory for a sliced parallax viewport
<code>gs_free_svpv</code>	Free memory for a sliced parallax viewport

Animation Functions

<u>Function</u>	<u>Description</u>
<code>gs_load_anim</code>	Load an anim object from disk
<code>gs_free_anim</code>	Free an anim object from memory
<code>gs_free_cplx</code>	Free a complex anim object from memory
<code>gs_get_display_list</code>	Allocate a display list
<code>gs_free_display_list</code>	Free a display list
<code>gs_init_anim</code>	Initialize a display list for use
<code>gs_rs_offset</code>	Specify real space X and Y offsets within virtual space
<code>gs_rs_window</code>	Shift a real space window which is smaller than the bitmap area to any position over the bitmap
<code>gs_set_anim_bounds</code>	Specify the legal bounds within which anim objects may roam
<code>gs_disable_anim_bounds</code>	Turn off bounds checking
<code>gs_enable_anim_bounds</code>	Turn on bounds checking
<code>gs_next_anim_page</code>	Tell the animation system to draw into the other bitmap of a double buffered display
<code>gs_get_anim_save_area</code>	Allocate memory for saving background data
<code>gs_get_parent_save_area</code>	Allocate background save memory for a parent object and all children
<code>gs_free_anim_save_area</code>	Release save area memory
<code>gs_free_parent_save_area</code>	Release save area memory for a parent object and all attached children
<code>gs_add_anim</code>	Add an anim object to a display list
<code>gs_add_parent</code>	Add a parent anim and all children to a display list
<code>gs_remove_anim</code>	Remove an anim object from the animation system
<code>gs_sort_anims</code>	Sort all anim objects in a display list
<code>gs_draw_anims</code>	Copy all anim objects in both display lists to the current drawing bitmap
<code>gs_restore_anim_bg</code>	Remove all anim objects from the current drawing bitmap
<code>gs_draw_anim_objs</code>	Copy all anim objects in both display lists to the current drawing bitmap after first calling <code>gs_restore_anim_bg</code>
<code>gs_reverse_anim</code>	Reverse an object's animation direction
<code>gs_anim_forward</code>	Set an object's animation direction to forward
<code>gs_anim_backward</code>	Set an object's animation direction to backward
<code>gs_clear_anim</code>	Make an active anim object invisible
<code>gs_enable_anim</code>	Make an invisible anim object visible again
<code>gs_set_anim_flicker</code>	Cause an anim object to flicker when displayed
<code>gs_clear_anim_flicker</code>	Stop an anim object from flickering
<code>gs_set_anim_merge</code>	Turn on translucency for an anim object
<code>gs_clear_anim_merge</code>	Turn off translucency for an anim object
<code>gs_enable_anim_collision</code>	Allow an anim object to collide with other anim objects
<code>gs_disable_anim_collision</code>	Disallow an anim object to collide with other anim objects
<code>gs_enable_anim_collision_bg</code>	Allow an anim object to collide with background data
<code>gs_disable_anim_collision_bg</code>	Disallow an anim object to collide with background data
<code>gs_set_anim_cell</code>	Set an object's current animation frame

<code>gs_move_anim</code>	Move an anim object to an X,Y coordinate in a bitmap
<code>gs_anim_obj</code>	Move and animate an anim object
<code>gs_anim_parent</code>	Move and animate a parent anim and all attached children
<code>gs_anim_children</code>	Move a parent anim, but only animate the child anims
<code>gs_set_collision</code>	Supply a collision handler for object-to-object collisions from a display list
<code>gs_clear_collision</code>	Disable collision handling for object-to-object collisions from a display list
<code>gs_set_collision_bg</code>	Supply a collision handler for object-tobackground collisions from a display list
<code>gs_clear_collision_bg</code>	Disable collision handling for object-tobackground collisions from a display list
<code>gs_add_anim_cplx</code>	Add a complex anim object to a display list
<code>gs_remove_cplx</code>	Remove an anim complex from the animation system
<code>gs_get_cplx_save_area</code>	Allocate a save area for background graphics
<code>gs_free_cplx_save_area</code>	Release save area memory
<code>gs_set_cplx_prio</code>	Set the display priority for an anim complex
<code>gs_set_cplx_seq</code>	Set the current animation sequence for an anim complex
<code>gs_set_cplx_cell</code>	Set the current animation frame for the current anim sequence of an main complex
<code>gs_set_cplx_flicker</code>	Cause an anim complex to flicker when displayed
<code>gs_clear_cplx_flicker</code>	Stop an anim complex from flickering
<code>gs_set_cplx_merge</code>	Turn on translucency for an anim complex
<code>gs_clear_cplx_merge</code>	Turn off translucency for an anim complex
<code>gs_clear_cplx</code>	Make an active main complex invisible
<code>gs_enable_cplx</code>	Make an invisible anim complex visible again
<code>gs_reverse_cplx</code>	Reverse an anim complex's animation direction
<code>gs_cplx_forward</code>	Set an anim complex's animation direction to forward
<code>gs_cplx_backward</code>	Set an anim complex's animation direction to backward
<code>gs_enable_cplx_collision</code>	Allow an anim complex to collide with other objects in the same display list
<code>gs_disable_cplx_collision</code>	Disable an anim complex from colliding with other objects in the same display list
<code>gs_enable_cplx_collision_bg</code>	Allow an anim complex to collide with background data
<code>gs_disable_cplx_collision_bg</code>	Disable an anim complex from colliding with background data
<code>gs_move_cplx</code>	Move an anim complex to an X,Y coordinate in a bitmap
<code>gs_anim_cplx</code>	Move and animate the current animation sequence of an anim complex
<code>gs_anim_cplx_parent</code>	Move and animate the current animation sequence of an anim complex and all children
<code>gs_anim_cplx_children</code>	Move an anim complex, but only animate attached child anims
<code>gs_re_dim</code>	Specify a smaller real space than the entire bitmap area
<code>gs_add_sprite</code>	Instructs the display system to use your sprite data for the specified sprite number
<code>gs_move_sprite</code>	moves a sprite to specified position
<code>gs_clear_sprite</code>	make sprite invisible

Sound Functions

<u>Function</u>	<u>Description</u>
gs_open_sound	Open the interrupt driven sound system
gs_close_sound	Close the sound system
gs_start_sound	Play a sound sample through a channel
gs_stop_sound	Stop any sounds on a channel
gs_set_volume	Change the volume of a sound playing
gs_set_period	Change the period of a sound playing
gs_sound_check	Check the status of the sound channels
gs_load_raw_sound	Load a file from disk as raw sound data
gs_load_iff_sound	Load an IFF 8SVX sound file from disk
gs_free_sound	Release sound sample data

Utility Functions

<u>Function</u>	<u>Description</u>
gs_open_libs	Open specified AmigaDOS ROM libraries
gs_close_libs	Close all opened AmigaDOS libraries
gs_random	Quickly generate a 16 bit random number
gs_joystick	Poll a joystick port
gs_task_priority	Set the programs task priority
gs_file_requester	Prompt the user for a path and file name
gs_LEDon	Turn the audio filter on
gs_LEDOff	Turn the audio filter off
gs_version	Return GameSmith.lib revision number
gs_add_vb_server	Add a function to the vertical blank server chain at the given priority
gs_remove_vb_server	Remove a function from the vertical blank server chain
gs_open_vb_timer	Open and initialize the GameSmith vertical blank interrupt timer
gs_close_vb_timer	Close the vertical blank timer
gs_vb_time	Get the count value from the vertical blank timer
gs_vb_timer_reset	Reset the timer clock count to zero
gs_init_vector	Initialize a vector for use
gs_step_vector	Get next vector point using the specified gradient
gs_ustep_vector	Get next vector point using the specified gradient, but with more uniform movement

11 GameSmith Library Reference

This chapter describes all GameSmith library functions in detail. Functions are listed in alphabetical order, and contain the following segments (when applicable):

<i>Function</i>	Shows the function name and purpose.
<i>Synopsis</i>	Shows the declaration of the function in ‘C’ form. Return values and input parameters are shown. Assembly registers are also shown.
<i>Description:</i>	Describes what the function does, and how to use it. All information needed to use the function is presented. Where necessary, technical details on how the function operates are also discussed here.
<i>Returns</i>	Any return values from the function is described here. Error codes are presented in this segment.
<i>Example</i>	This segment contains example code in ‘C’ that demonstrates use of the function. Additional example code can be found in the example disks provided with the system.
<i>See also</i>	Refers you to other, related functions. Reading about related functions will give you a better overall understanding of the GameSmith system and how to use its library functions.

<i>Function</i>	gs_add_anim Add an anim object to a display list.
<i>Synopsis</i>	<pre>status = gs_add_anim(handle,anim, x, y); d0 d0 a0 d1 d2 int handle; /* display list handle */ int status; /* status return */ struct anim struct *anim; /*ptr to anim struct*/ int x,y; /* X,Y coordinates for object placement */</pre>

Libraries none

Description Before an anim object can be displayed in a bitmap, it must first be added to a display list so the animation system knows about it. Objects are inserted into a display list according to priority settings (the prio field of the anim structure), and given an active status. Active anims will be copied to a bitmap in the appropriate manner with the next call to `gs_draw_anims`. Any background save areas that need to be allocated are allocated. The X and Y coordinates are saved in the anim structure for object placement Remember that the X,Y coordinates refer to the upper left hand corner of the object, and may fall anywhere within the 32K virtual space range.

By default, the first cell in the anim sequence will be used for display. Also, the animation direction is reset to forward, and the INVISIBLE and FLICKER flags are cleared.

All functions that add objects to a display list can take some time. Because of this, it is not recommended that your program do a lot of adding and removing of objects to and from a display list during high activity periods. See the functions `gs_clear_anim` and `gs_enable_anim` instead if you need to do a lot of adding and removing.

<i>Returns</i>	A successful completion is indicated by a return of zero. Possible errors are: 1 not enough Chip RAM for save area(s) 2 anim already active in display system
<i>See also</i>	<code>gs_add_parent, gs_get_anim save_area, gs_remove_anim, gs_set_anim_cell, gs_get_display_list</code>
<i>Function</i>	gs_add_anim_cplx Add an anim complex to a display list
<i>Synopsis</i>	<pre> st=gs_add_anim_cplx(handle,cplx, x, y,seq, prio); d0 d0 a0 d1 d2 d3 d4 int handle; /* display list handle */ int status; /* status return */ struct anim_cplx *cplx; /* ptr to anim complex */ int x,y; /*X,Y coordinates for object placement*/ int seq; /* starting animation sequence(0-n)*/ int prio; /* display priority for object */ </pre>
<i>Libraries</i>	none
<i>Description</i>	Before an anim object can be displayed in a bitmap, it must first be added to a display list so the animation system knows about it. Objects are inserted into a display list according to display priority, and given an active status. Active anims will be copied to a bitmap (using the active anim sequence, where anim complexes are concerned) in the appropriate manner with the next call to <code>gs_draw_anims</code>. This function adds the specified anim complex to a display list along with any attached anims. Any background save areas that need to be allocated are allocated. The X and Y coordinates are saved in the anim structure for object placement. Remember that the X,Y coordinates refer to the upper left hand corner of the object, and may fall anywhere within the 32K virtual space range. This applies to parent objects only, since attached anims are placed relative to the parent. Note that this function adds the specified complex and all attached anims (if any) to the display list. By default, the first cell in the specified anim sequence will be used for display. Also, the animation direction is reset to forward, and the INVISIBLE and FLICKER flags are cleared. All functions that add objects to an animation display list can take some time. Because of this, it is not recommended that your program do a lot of adding and removing of objects to and from a list during high activity periods. See the functions <code>gs_clear_cplx</code> and <code>gs_enable_cplx</code> instead if you need to do a lot of adding and removing.
<i>Returns</i>	A successful completion is indicated by a return of zero. Possible errors are: 1 not enough Chip RAM for save area(s) 3 anim complex already active in display system 4 specified sequence number greater than last in complex
<i>See also</i>	<code>gs_get_cplx_save_area, gs_remove_cplx, gs_free_cplx_save_area, gs_set_cplx_cell, gs_get_display_list</code>

<i>Function</i>	gs_add_parent Add a parent anim and all attached children to a display list
<i>Synopsis</i>	<pre> status = gs_add_parent(handle, anim, x, y); d0 d0 a0 d1 d2 int handle; /* display list handle */ int status; /* status return */ struct anim_struct *anim; /*ptr to parent anim */ int x,y; /* X,Y coordinates for object placement */ </pre>
<i>Libraries</i>	none
<i>Description</i>	Before an anim object can be displayed in a bitmap, it must first be added to a display list so the animation system knows about it. Objects are inserted into a display list according to priority settings, and given an active status. Active anims will be copied to a bitmap in the appropriate manner with the next call to <code>gs_draw_anims</code>. This function adds the specified anim object to a display list along with any attached anims. Any background save areas that need to be allocated are allocated. The X and Y coordinates are saved in the anim structure for object placement. Remember that the X,Y coordinates refer to the upper left hand corner of the object, and may fall anywhere within the 32K virtual space range. This applies to the parent object only, since attached anims are placed relative to the parent. All functions that add objects to a display list can take some time. Because of this, it is not recommended that your program do a lot of adding and removing of objects to and from a display list during high activity periods. See the functions <code>gs_clear_anim</code> and <code>gs_enable_anim</code> instead if you need to do a lot of adding and removing.
<i>Returns</i>	A successful completion is indicated by a return of zero. Possible errors are: 1 not enough Chip RAM for save area(s) 2 at least one anim already active in display system In the event of an error, the parent anim and all attached anims are removed from the display list, and removed from active status. Note that any save areas that were successfully allocated prior to causing a code 1 return are NOT freed upon return.
<i>See also</i>	<code>gs_add_anim</code> , <code>gs_get_parent_save_area</code> , <code>gs_remove_anim</code> , <code>gs_free_parent_save_area</code> , <code>gs_get_display_list</code>
<i>Function</i>	gs_add_sprite Instructs the display system to use your sprite data for the specified sprite number
<i>Synopsis</i>	<pre> void gs_add_sprite(d, sprite, data, height, x, y); struct display_struct *d; /* display structure in use */ int sprite; /* sprite number from 0 - 7 */ unsigned short *data; /* pointer to actual sprite data bank */ int height; /* sprite height in scan lines */ int x,y; /* X & Y screen position */ </pre>

Description This function tells the display system to use your sprite data for the specified sprite number, and to position it at the given X and Y coordinates on the screen. The sprite will show up after the next vertical blank interrupt.

NOTE: since sprite position is maintained in the first two words of the data itself, you must have separate data banks for each, sprite. Remember that the X and Y coordinates are relative to the upper left hand corner of the sprite.

See also `gs_move_sprite`, `gs_clear_sprite`

Function **gs_add_vb_server**
Add a function to the system vertical blank server chain

Synopsis

```
server = gs_add_vb_server(funcnt ,priority);
d0                      a0      d0

struct Interrupt *server; /* return reference structure */
void *funcnt; /* pointer to function to add */
int priority; /* server priority */
```

Libraries none

Description This function provides a high level interface to the system's vertical blank server chain. It allows you to place any function in the vertical blank. Priorities range from -128 to 127 (8 bit value). Unless you need your function to process before or after another function in the server chain, use a priority of 0.

There are only a few restrictions. Vertical blank server functions accept no input, and any return value is simply thrown away. Also, the user should make every effort to make the function as short and efficient as possible, as other things need to be done in the vertical blank interrupt. 'C' programmers must make sure that the interrupt code does not do stack checking. SAS users should specify the `_interrupt` keyword when declaring the function. Other than that, the programmer is free to use all system registers.

Returns If the function is successfully added, a pointer to an Interrupt structure is returned. This pointer value must be passed to `gs_remove_vb_server` before your program exits. You need do nothing else with it. In the unlikely event that the function could not be added, NULL is returned.

Example

```
/* Duplicate the gs_vb_timer function. */
void timer(void);
struct Interrupt *server;
unsigned long time=0;

void timer() {
    time++; /* count vertical blank interrupts */
}
main ( )
{
    if (!(server=gs_add_vb_server(&timer,0)))
    {
        printf("\nUnable to add timer to vertical blank\n");
        exit(-1);
    }

    /* wait until user presses left mouse button
    while (!(gs_joystick(0) & JOY_BUTTON1));
```



```
gs_remove_vb_server(server);
printf("\n%d vertical blank periods were timed\n",time);
```

See also gs_remove_vb_server

Function **gs_alloc_spvp**
allocate memory for a sliced parallax viewport

Synopsis status = gs_alloc_spvp(spvp);

struct gs_spvp *spvp; /* pointer to initialized spvp structure */

Returns 0 memory allocated OK
-1 Insufficient Memory

Libraries None

See also gs_free_spvp

Function **gs_anim_backward**
Set the animation direction for an anim object to backward

Synopsis void gs_anim_backward(anim) ;
a0

struct anim struct *anim; /* pointer to anim structure */

Description This function sets the animation direction of an anim object to backward. Backward direction means that when animating an object, the cell number will decrease from frame to frame. That is, from cell 5, the next frame will be cell 4, and so on to the end of the sequence. The next call to a function which animates the object will reflect the animation direction.

Libraries None

See also gs_anim_forward, gs_anim_obj, gs_anim_parent, gs_anim_children

Function **gs_anim_children**
Reposition an anim object and animate all attached child anims

Synopsis void gs_anim_children(anim, x, y);
a0 d0 d1

struct anim struct *anim; /* pointer to anim structure */
int x,y; /* X,Y coordinates for object placement */

Libraries None

Description This function repositions an anim object and all attached anims at the specified x,y coordinates within virtual space, but only animates the child anims of the parent. Valid virtual space values range from 0 to 32767 (0x7fff hex). The specified parent anim is moved to the new coordinates, but not animated. Note that no actual drawing takes place as a result of this call. Depending on each child object's animation direction, the next or previous cell in the sequence is readied for display. In the case of single shot (non-

repeating) anims (forward and/or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. As always, the placement of child anims is relative to the placement of the parent.

X and Y coordinates refer to the upper left hand corner of the parent image.

See also `gs_anim_parent`, `gs_draw_anims`, `gs_set_anim_bounds`

Function **gs_anim_cplx**
Reposition and animate an anim complex

Synopsis

```
void gs_anim_cplx(cplx, x, y) ;
                a0 d0 d1

struct anim cplx *cplx; /* pointer to anim complex */
int x,y;                /* X,Y coordinates for object placement */
```

Description This function repositions and animates an anim complex at the specified x,y coordinates within virtual space. Valid virtual space values range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. Depending on the object's animation direction, the next or previous cell in the current sequence is readied for display. In the case of single shot (non-repeating) anims (forward and/ or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. Movement of a parent complex also moves all attached child anims, although the child anims are not animated by this function.

X and Y coordinates refer to the upper left hand corner of an image.

See also `gs_move_cplx`, `gs_draw_anims` , `gs_set_anim_bounds` , `gs_anim_cplx_parent`

Function **gs_anim_cplx_children**
Reposition an anim complex and animate all attached child anims

Synopsis

```
void gs_anim_cplx_children(cplx, x, y);
                a0 d0 d1

struct anim cplx *cplx; /* pointer to anim complex */
int x,y;                /* X,Y coordinates for object placement */
```

Libraries none

Description This function repositions an anim complex and all attached anims at the specified x,y coordinates within virtual space, but only animates the child anims of the parent complex. Valid virtual space values range from 0 to 32767 (0x7fff hex). The specified parent complex is moved to the new coordinates, but not animated. Note that no actual drawing takes place as a result of this call. Depending on each child object's animation direction, the next or previous cell in the sequence is readied for display. In the case of single shot (non-repeating) anims (forward and/or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. As always, the placement of child anims is relative to the placement of the parent.

X and Y coordinates refer to the upper left hand corner of the parent complex.

See also `gs_anim_cplx_parent`, `gs_draw_anims`, `gs_set_anim bounds`

<i>Function</i>	gs_anim_cplx_parent Reposition and animate an anim complex and all attached child anims
<i>Synopsis</i>	<pre>void gs_anim cplx_parent(cplx, x, y); a0 d0 d1 struct anim cplx *cplx; /* pointer to anim complex */ int x,y; /* X,Y coordinates for object placement */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function repositions and animates an anim complex and all attached anims at the specified x,y coordinates within virtual space. Valid virtual space values range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. Depending on each object's animation direction, the next or previous cell in the sequence is readied for display. In the case of single shot (non-repeating) anims (forward and/or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. As always, the placement of child anims is relative to the placement of the parent. X and Y coordinates refer to the upper left hand corner of the parent image.
<i>See also</i>	gs_anim_cplx, gs_draw_anims, gs_set_anim_bounds
<i>Function</i>	gs_anim_forward Set the animation direction for an anim object to forward
<i>Synopsis</i>	<pre>void gs_anim forward(anim); a0 struct anim_struct *anim; /* pointer to anim structure */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function sets the animation direction of an anim object to forward. Forward direction means that when animating an object, the cell number will increase from frame to frame. That is, from cell 1, the next frame will be cell 2, and so on to the end of the sequence. The next call to a function which animates the object will reflect the animation direction.
<i>See also</i>	gs_anim_backward, gs_anim_obj, gs_anim_parent, gs_anim children
<i>Function</i>	gs_anim_obj Reposition and animate an anim object
<i>Synopsis</i>	<pre>void gs_anim_obj(anim, x, y); a0 d0 d1 struct anim struct *anim; /* pointer to anim structure */ int x,y; /* X,Y coordinates for object placement */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function repositions and animates an anim object at the specified x,y coordinates within virtual space. Valid virtual space values range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. Depending on the object's

animation direction, the next or previous cell in the sequence is readied for display. In the case of single shot (non-repeating) anims (forward and/ or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. Movement of a parent anim also moves all attached child anims, although the child anims are not animated by this function.

X and Y coordinates refer to the upper left hand corner of an image.

See also `gs_move_anim`, `gs_draw_anims`, `gs_set_anim_bounds`, `gs_anim_parent`

Function **gs_anim_parent**
Reposition and animate an anim object and all attached child anims

Synopsis

```
void gs_anim parent(anim, x, y) ;
                    a0  d0 d1

struct anim struct *anim; /* pointer to anim structure */
int x,y; /* X,Y coordinates for object placement */
```

Libraries None

Description This function repositions and animates an anim object and all attached anims at the specified x,y coordinates within virtual space. Valid virtual space coordinate values range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. Depending on each object's animation direction, the next or previous cell in the sequence is readied for display. In the case of single shot (non-repeating) anims (forward and/or backward), this function has no effect on the current cell when the end of the sequence is reached. The same cell is simply redisplayed. As always, the placement of child anims is relative to the placement of the parent.

X and Y coordinates refer to the upper left hand corner of the parent image.

See also `gs_anim_obj`, `gs_draw_anims`, `gs_set_anim_bounds`

Function **gs_blit_clear**
Clear an image from a bitmap using the blitter

Synopsis

```
void gs_blit_clear(image, bitmap, x, y);
                    a0    a1    d0 d1

struct blit_struct *image; /* ptr to blitter image struct */
struct BitMap *bitmap; /* ptr to bitmap to draw into */
int x,y; /* X,Y offsets into bitmap for upper left hand
corner of image*/
```

Libraries GRAPHICS

Description This function clears image data from the specified bitmap using the "cookie cut" blitter minterm. This requires that the mask field of the blitter structure be properly filled. Basically speaking, the mask acts like a cookie cutter pattern, or a paper doll outline, and defines the area to be cleared from the destination bitmap. The image mask is a logical OR of all bitplanes in the image. For a more complete explanation of blitter minterms and mask usage, see the AMIGA ROM Kernel Reference Manual.: Libraries, or the AMIGA Hardware Reference Manual. Note that blitter structures created by CITAS automatically

have the mask field properly filled.

The area of the bitmap in the exact shape of the image mask is cleared to zero or background color. This is useful in animation, for instance, when butting images to a bitmap that has no background graphics (a blank background). In this case the background does not have to be saved, and each image can simply be removed from the screen by this function before moving it to another position in the bitmap. This two step process is much faster than the three step process used when background graphics are involved.

The X and Y coordinates are offsets into the bitmap for clearing the image, and correspond to the upper left hand corner of the image.

For speed, no bounds checking is done. It is up to the programmer to insure that the image will fit inside the bounds of the bitmap.

See also `gs_blit_image`

Function **gs_blit_copy**
Copy image data to a bitmap using the blitter

Synopsis

```
void gs_blit_copy(image, bitmap, x, y);
                   a0      a1      d0 d1

struct blit_struct *image; /* pointer to blitter image struct */
struct BitMap *bitmap;    /* ptr to bitmap to draw into */
int x,y;                  /* X & Y offsets into bitmap for
                           upper left hand corner of image */
```

Libraries **GRAPHICS**

Description This function copies image data into the specified bitmap. Unlike `gs_blit_image`, however, it does NOT use an image mask. This is a straight copy of the image data into the destination bitmap. Also, no background graphics are saved by this function, even if the save field is nonzero.

This function is ideal for blitting graphics that are an even multiple of 16 bits wide. And since it does not use an image mask, it is much faster than `gs_blit_image`.

The X and Y coordinates are offsets into the bitmap for placement of the image, and correspond to the upper left hand corner of the image.

For speed, no bounds checking is done on the image data. It is up to the programmer to insure that the image will fit inside the bounds of the bitmap.

See also `gs_blit_image`, `gs_blit_save_bg`, `gs_blit_restore`

Function **gs_blit_copy_fine**
Copy image data to a bitmap using the blitter

Synopsis

```
void gs_blit_copy_fine
(image, bitmap, x, y, mod, mask1, mask2);
    a0      a1      d0 d1 d2      d3      d4

struct blit_struct *image; /* ptr to blitter image struct */
struct BitMap *bitmap; /* ptr to bitmap to draw into */
```

```
int x,y; /* X & Y offsets into bitmap for upper left hand corner of image
*/
int mod; /* source modulo */
int mask1; /* 1st word mask */
int mask2; /* last word mask */
```

Libraries GRAPHICS

Description This function is identical to the function `gs_blit_copy` except for the additional of the last three parameters, which allows finer control over how an image is copied. `mod` allows specification of a modulo for the source data. A modulo is the amount of data per line that should be skipped before starting on the next line of data. In this case, the modulo specifies the number of BYTES that should be skipped each line of source data. Note that the width field of the blitter structure specifies the width of each source line in WORDS. The width and the modulo must add up to the total width of each line of source data. E.g. by decreasing the width by 1, you need to increase the modulo by 2. This allows clipping images on word boundaries.

The `mask1`, and `mask2` parameters allow masking of the 1st word (16 bits) of source data per line and the last word of source data per line respectively. The masks are bit masks, and any bit turned on in the mask allows data to be copied in that bit position. For instance, if the 1st word mask were 0x00ff, only the lower 8 bits of the 1st data word would actually be copied. This allows clipping images on pixel boundaries.

Note that although `mod`, `mask1` and `mask2` are 32 bit integers, only the lower 16 bits are actually used. This is for the sake of consistency with all other GameSmith function calls from 'C'. Assembler programmers need only fill the appropriate registers with word data.

See also `gs_blit_image_fine`, `gs_blit_copy`, `gs_blit_clear`

Function **gs_blit_copy2**
Copy image data to a bitmap using the blitter

Synopsis `void gs_blit_copy2(image,bitmap);`
 a0 a1

`struct blit_struct *image; /* ptr to blitter image struct */`
`struct BitMap *bitmap; /* ptr to bitmap to draw into */`

Libraries GRAPHICS

Description This function is identical to `gs_blit_copy` except that it relies on key fields to have already been computed from a call to `gs_blit_save_bg`. You **MUST** call `gs_blit_save_bg` first if you want to call this function. This function is used primarily by the animation system as a means to speed blitter operations to the screen in the event that background graphics data is being saved.

Note the absence of an X,Y coordinate for image placement. This is because bitmap position has already been computed by `gs_blit_save_bg`.

See also `gs_blit_copy`, `gs_blit_save_bg`, `gs_blit_restore`

Function **gs_blit_fill**
Fill a Chip memory area with a value using the blitter

<i>Synopsis</i>	<pre>void gs_blit_fill(ushort *start,int words,int fill); a0 d0 d1 unsigned short start; /* pointer to the 1st word to be filled */ int words; /* the number of words (16 bit locations) to fill */ int fill; /* 16 bit values to stuff in the memory area. Note that although a 32 bit value is passed from 'C', only the lower 16 bits are used. The upper 16 bits should always be set to 0 */</pre>
<i>Description</i>	This function uses the blitter to fill a Chip memory area with a given word pattern. Speed tests show this function to actually be quite a bit faster than the system BltClear(). There is no CPU version of this function for obvious reasons.
<i>Function</i>	gs_blit_free_save_area Free a background data save area
<i>Synopsis</i>	<pre>void gs_blit_free_save_area(image); struct blit_struct *image; /* pointer to blitter image struct */</pre>
<i>Description</i>	This function frees the memory buffer previously allocated by <code>gs_blit_get_save_area</code> for the image. Any program that allocates a save area should free it before exiting.
<i>See also</i>	<code>gs_blit_get_save_area</code>
<i>Function</i>	gs_blit_get_save_area Get a save area for background data
<i>Synopsis</i>	<pre>status = gs_blit_get_save_area(image); a0 a0 int status; /* return */ struct blit_struct *image; /* pointer to blitter image struct */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function allocates a memory buffer in Chip RAM which is the same size as the image data. This buffer is used to save background graphics data when blitting in the ANIM_SAVE_BG mode. If successful, this function fills the save field of the blitter structure with a pointer to the allocated buffer.
<i>Returns</i>	A value of zero is returned if successful. Otherwise a nonzero value indicates that there was insufficient Chip RAM from which to allocate the save area. As stated above, the save field of the blitter structure points to the allocated buffer if successful.
<i>See also</i>	<code>gs_blit_free_save_area</code>
<i>Function</i>	gs_blit_image Copy image data to a bitmap using the blitter
<i>Synopsis</i>	<pre>void gs_blit_image(image, bitmap, x, y); a0 a1 d0 d1 struct blit_struct *image; /* pointer to blitter image struct */ struct BitMap *bitmap; /* pointer to bitmap to draw into */ int x,y; /* X & Y offsets into bitmap for upper left hand corner of image */</pre>

Libraries **GRAPHICS**

Description This function copies image data into the specified bitmap using the "cookie cut" blitter minterm. This requires that the mask field of the blitter structure be properly filled. Basically speaking, the mask acts like a cookie cutter pattern, or a paper doll outline, and defines how the image data is copied into the destination bitmap. The image mask is a logical OR of all bitplanes in the image. For a more complete explanation of blitter minterms and mask usage, see the AMIGA ROM Kernel Reference Manual: Libraries, or the AMIGA Hardware Reference Manual. Note that blitter structures created by CITAS automatically have the mask field properly filled.

If the save field of the blitter structure is nonzero, it is assumed to point to a valid save area, and background graphics are copied to the save area before the image is copied into the bitmap.

The X and Y coordinates are offsets into the bitmap for placement of the image, and correspond to the upper left hand corner of the image.

For speed, no bounds checking is done on the image data. It is up to the programmer to insure that the image will fit inside the bounds of the bitmap.

This function, as with all of the GameSmith blitter functions, calls the system graphics library functions OwnBlitter and DisownBlitter to insure compatibility with the rest of the Amiga operating system, and to allow programs which use the GameSmith functions to multitask.

See also `gs_blit_image_fine`, `gs_blit_copy`, `gs_blit_clear`,
`gs_blit_get_save_area`, `gs_blit_restore`

Function **gs_blit_image_fine**
Copy image data to a bitmap using the blitter

Synopsis `void gs_blit_image_fine(image, bitmap, x, y, mod, mask1, mask2);`
 a0 a1 d0 d1 d2 d3 d4

```
struct blit_struct *image; /* pointer to blitter image struct */
struct BitMap *bitmap; /* pointer to bitmap to draw into */
int x,y;        /* X & Y offsets into bitmap for upper left hand corner of
image */
int mod;        /* source modulo */
int mask1;      /* 1st word mask */
int mask2;      /* last word mask */
```

Libraries **GRAPHICS**

Description This function is identical to the function `gs_blit_image` except for the additional of the last three parameters, which allows finer control over how an image is copied. `mod` allows specification of a modulo for the source data. A modulo is the amount of data per line that should be skipped before starting on the next line of data. In this case, the modulo specifies the number of BYTES that should be skipped each line of source data. Note that the width field of the blitter structure specifies the width of each source line in WORDS. The width and the modulo must add up to the total width of each line of source data. E.g. by decreasing the width by 1, you need to increase the modulo by 2. This allows clipping images on word boundaries.

The mask1, and mask2 parameters allow masking of the 1st word (16 bits) of source data per line and the last word of source data per line respectively. The masks are bit masks, and any bit turned on in the mask allows data to be copied in that bit position. For instance, if the 1st word mask were 0x00ff, only the lower 8 bits of the 1st data word would actually be copied. This allows clipping images on pixel boundaries.

Note that although mod, mask1, and mask2 are 32 bit integers, only the lower 16 bits are actually used. This is for the sake of consistency with all other GameSmith function calls from 'C'. Assembler programmers need only fill the appropriate registers with word data.

See also `gs_blit_image, gs_blit_copy, gs_blit_copy_fine`

Function **gs_blit_image2**
Copy image data to a bitmap using the blitter

Synopsis

```
void gs_blit_image2(image,bitmap);
                   a0      a1

struct blit_struct *image; /* pointer to blitter image struct */
struct BitMap *bitmap;    /* pointer to bitmap to draw into */
```

Libraries GRAPHICS

Description This function is identical to `gs_blit_image` except that it relies on key fields to have already been computed from a call to `gs_blit_save_bg`. You **MUST** call `gs_blit_save_bg` first if you want to call this function. This function is used primarily by the animation system as a means to speed blitter operations to the screen in the event that background graphics data is being saved.

Note the absence of an X,Y coordinate for image placement. This is because bitmap position has already been computed by `gs_blit_save_bg`.

See also `gs_blit_image, gs_blit_save_bg`

Function **gs_blit_memcpy**
Copy a block of Chip memory using the blitter

Synopsis

```
void gs_blit_memcpy(source,dest,words);
                   a0      a1      d0

void *source; /* address of memory block to copy*/
void *dest;   /* address of destination memory block */
int words;    /* number of 16 bit words to copy */
```

Libraries GRAPHICS

Description This function copies a linear block of memory in Chip RAM to another block of memory in Chip RAM using the blitter. Since the blitter only works with data in quantities of 16 bits (one word), only an even number of bytes can be copied.

Older blitter chips could not blit as much data in one operation as newer blitter chips. This function places no restriction on the amount of data to be copied.

<i>Function</i>	gs_blit_restore Restore background graphics using the blitter
<i>Synopsis</i>	<pre>void gs_blit_restore(image,bitmap); a0 a1 struct blit_struct *image; /* pointer to blitter image struct */ struct BitMap *bitmap; /* pointer to bitmap to draw into */</pre>
<i>Library</i>	GRAPHICS
<i>Description</i>	This function copies the contents of the save area to the position in the bitmap previously set up by a call to <code>gs_blit_image</code> , <code>gs_blit_save_bg</code> , or <code>gs_blit_copy</code> .
<i>See also</i>	<code>gs_blit_image</code> , <code>gs_blit_save_bg</code> , <code>gs_blit_copy</code> , <code>gs_blit_get_save_area</code>
<i>Function</i>	gs_blit_save_bg Save background graphics using the blitter
<i>Synopsis</i>	<pre>void gs_blit_save_bg(image,bitmap, x, y); a0 a1 d0 d1 struct blit_struct *image; /* pointer to blitter image struct */ struct BitMap *bitmap; /* pointer to bitmap to save from */ int x,y; /* X & Y offsets into bitmap for upper left hand corner of image */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function saves the background graphics from a bitmap into a save area so that images appear to move "on top of" the background through a process of save background, copy image, restore background. Needless to say, a save area must have previously been allocated for this function to work (see <code>gs_blit_get_save_area</code>). This function uses the blitter for the save process, and simply copies the background graphics to the save area. No image mask field need be set up for this operation. The X and Y coordinates are offsets into the bitmap for placement of the image, and correspond to the upper left hand corner of the image.
<i>See also</i>	<code>gs_blit_image2</code> , <code>gs_blit_copy2</code> , <code>gs_blit_get_save_area</code> , <code>gs_blit_restore</code>
<i>Function</i>	gs_chiprev Inquire chipset version
<i>Synopsis</i>	<pre>int gs_chiprev(void);</pre>
<i>Description</i>	This functions returns the chipset under which your program is running.
<i>Returns</i>	AGA_CHIPREV for the AGA chipset ECS_CHIPREV for the Enhanced Chip Set OCS_CHIPREV for the Original Chip Set
<i>Function</i>	gs_clear_anim Make an active anim object invisible

<i>Synopsis</i>	<pre>void gs_clear_anim(anim); a0 struct anim struct *anim; /* pointer to anim structure */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function makes an active anim object and all of its attached objects (if any) invisible. The objects remain active in the display lists, but simply do not interact with other objects or background data while invisible. An invisible object cannot collide with anything. The process is not immediate, however. In a single buffered environment, the next call to <code>gs_draw_anims</code> will make it invisible. In a double buffered environment, two calls to <code>gs_draw_anims</code> will be required to totally clear the object(s). The call(s) to <code>gs_draw_anims</code> are necessary for cleanup operations such as restoring the background data beneath the object(s) (if required). The anim object must be active in the animation system in order for this function to take affect.
<i>See also</i>	<code>gs_enable_anim</code> , <code>gs_move_anim</code> , <code>gs_anim_obj</code>
<i>Function</i>	gs_clear_anim_flicker Stop an active anim object from flickering
<i>Synopsis</i>	<pre>void gs_clear_anim_flicker(anim); a0 struct anim struct *anim; /* pointer to anim struct */</pre>
<i>Description</i>	This function stops the flicker effect in an active anim object and all of its attached anims (if any). The objects are displayed normally again.
<i>See also</i>	<code>gs_set_anim_flicker</code>
<i>Function</i>	gs_clear_anim_merge Turn off merge mode for an anim object
<i>Synopsis</i>	<pre>void gs_clear_anim_merge(anim); a0 struct anim struct *anim; /* pointer to anim struct */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function turns off merge mode for an anim object and all of its attached anims (if any). This function merely clears the ANIM_MERGE flag in the anim structure(s).
<i>See also</i>	<code>gs_set_anim_merge</code>
<i>Function</i>	gs_clear_collision Remove any object-to-object collision handler for a display list

<i>Synopsis</i>	<pre>void gs_clear_collision(handle); d0 int handle; /* display list handle */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function tells the animation system to ignore collisions between objects in the display list associated with handle. This is the default. You need not call this function before exiting. your program. Nor do you have to call this function before setting up a new collision handler.
<i>See also</i>	gs_set_collision
<i>Function</i>	gs_clear_collision_bg Remove any object-to-background collision handler for a display list
<i>Synopsis</i>	<pre>void gs_clear_collision_bg(handle) d0 int handle; /* display list handle */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function tells the animation system to ignore collisions with background data for objects in the display list associated with handle. This is the default. You need not call this function before exiting your program. Nor do you have to call this function before setting up a new collision handler.
<i>See also</i>	gs_set_collision_bg
<i>Function</i>	gs_clear_cplx Make an active anim complex invisible
<i>Synopsis</i>	<pre>struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function makes an active anim complex and all of its attached anims (if any) invisible. The objects remain active in the display lists, but simply no not interact with other objects or background data while invisible. An invisible object cannot collide with anything. The process is not immediate, however. In a single buffered environment, the next call to <code>gs_draw_anims</code> will make it invisible. In a double buffered environment, two calls to <code>gs_draw_anims</code> will be required to totally clear the object(s). The call(s) to <code>gs_draw_anims</code> are necessary for cleanup operations such as restoring the background data beneath the object(s) (if required). The anim complex must be active in the animation system in order for this function to take affect. An invisible object becomes visible again by moving it, animating it, or reenabling it.

<i>See also</i>	<code>gs_enable_cplx</code> , <code>gs_move_cplx</code> , <code>gs_anim_cplx</code>
<i>Function</i>	gs_clear_cplx_flicker Stop an active anim complex from flickering
<i>Synopsis</i>	<pre>void gs_clear_cplx_flicker(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function stops the flicker effect in an active anim complex and all of its attached anims (if any). The objects are displayed normally again.
<i>See also</i>	<code>gs_set_cplx_merge</code>
<i>Function</i>	gs_clear_cplx_merge Turn off merge mode for an anim complex
<i>Synopsis</i>	<pre>void gs_clear_cplx_merge(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function turns off merge mode for all anims in an anim complex and all of its attached anims (if any). This function merely clears the ANIM_MERGE flag in the anim structures.
<i>See also</i>	<code>gs_set_cplx_merge</code>
<i>Function</i>	gs_clear_sprite Make sprite invisible
<i>Synopsis</i>	<pre>void gs_clear_sprite(d, sprite); struct display struct *d; /* display structure in use */ int sprite; /* sprite number from 0 - 7 */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function clears the specified sprite from the screen (makes it invisible).
<i>Function</i>	gs_close_libs Close all opened AmigaDOS libraries
<i>Synopsis</i>	<pre>void gs_close_libs(void);</pre>
<i>Libraries</i>	none
<i>Description</i>	This function closes all AmigaDOS libraries that were previously opened by one or more calls to <code>gs_open_libs</code> .

<i>See also</i>	<code>gs_open_libs</code>
<i>Function</i>	gs_close_sound Close the sound system
<i>Synopsis</i>	<pre>void gs_close_sound(void);</pre>
<i>Description</i>	This function closes the GameSmith sound system. The interrupt routines are removed, all buffers are freed, and if necessary, the audio.device is freed.
<i>See also</i>	<code>gs_open_sound</code>
<i>Function</i>	gs_close_vb_timer Close the custom vertical blank timer
<i>Synopsis</i>	<pre>void gs_close_vb_timer(void);</pre>
<i>Libraries</i>	none
<i>Description</i>	This function removes the custom timer routine, previously installed with <code>gs_open_vb_timer</code> , from the system's vertical blank interrupt server chain.
<i>See also</i>	<code>gs_open_vb_timer</code> , <code>gs_vb_time</code> , <code>gs_vb_timer_reset</code>
<i>Function</i>	gs_cplx_backward Set the animation direction for an anim complex to backward
<i>Synopsis</i>	<pre>void gs_cplx_backward(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function sets the animation direction of the current animation sequence of an anim complex to backward. Backward direction means that when animating an object, the cell number will decrease from frame to frame. That is, from cell 5, the next frame will be cell 4, and so on to the end of the sequence. The next call to a function which animates the object will reflect the animation direction. Note that changing the animation sequence will reset the animation direction to the default of forward.
<i>See also</i>	<code>gs_cplx_forward</code> , <code>gs_anim_cplx</code> , <code>gs_anim_cplx_parent</code> , <code>gs_anim_cplx_children</code>
<i>Function</i>	gs_cplx_forward Set the animation direction for an anim complex to forward
<i>Synopsis</i>	<pre>void gs_cplx_forward(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>

<i>Libraries</i>	none
<i>Description</i>	This function sets the animation direction of the current animation sequence of an anim complex to forward. Forward direction means that when animating an object, the cell number will increase from frame to frame. That is, from cell 1, the next frame will be cell 2, and so on to the end of the sequence. The next call to a function which animates the object will reflect the animation direction.
<i>See also</i>	<code>gs_cplx_backward</code> , <code>gs_anim_cplx</code> , <code>gs_anim_cplx_parent</code> , <code>gs_anim_cplx_children</code>
<i>Function</i>	gs_create_display Create a ViewPort display
<i>Synopsis</i>	<pre>status = gs_create_display(display); int status; struct display_struct *display; /* ptr to display structure */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function creates a custom display using the system's View and ViewPort graphics primitives. Because of this it is very fast at display update, particularly page flipping a double buffered display, and remains compatible with all Amiga models. But also because of this it takes over the Amiga's entire display, and is not compatible with the normal Intuition environment (although Intuition screens have no problem running behind the scenes). If you need Intuition compatibility, then you should use Intuition screens instead for your display. Once the Amiga operating system has set up a controlling copper list for the new display, the GameSmith display system takes over, making many custom modifications. This hybrid approach gives system compatibility, while allowing more advanced hardware controlling features like dual playfield and split screen synchronized scrolling at full display speeds. Unlike the simpler function <code>gs_display</code>, this function allows complete control over the Amiga display through the use of the GameSmith display and viewport structures. Refer to the User's Guide for complete information on the display and viewport structures. Note that although this function sets up the entire display, it will not be shown until you call <code>gs_show_display</code>. ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the address of the display structure be placed on the stack before making the call. If successful, zero is returned as the status. Otherwise a nonzero value is returned. Possible error codes are as follows: -1 Unable to allocate a bitmap -2 Unable to allocate a ViewExtra structure -3 Unable to allocate a ViewPortExtra structure -4 Unable to obtain a MonitorSpec from OpenMonitor -5 Unable to allocate a ColorMap

- 6 Unable to obtain a DisplayInfo record from modes
- 7 VideoControl failed to attach a ColorMap to a ViewPort
- 8 Unable to fill a MonitorInfo structure from GetDisplayInfoData
- 9 Unable to fill a DimensionInfo structure from GetDisplayInfoData
- 12 Unable to allocate system ViewPort structure
- 13 Unable to allocate system RasInfo structure
- 14 Unable to allocate system View structure

See also `gs_display`, `gs_show_display`, `gs_flip_display`, `gs_old_display`,
`gs_remove_display`, `gs_scan_copper`

Function **gs_disable_anim_bounds**
Turn off bounds checking for objects in a display list

Synopsis

```
void gs_disable_anim_bounds(handle);
                               d0

int handle; /* display list handle */
```

Libraries none

Description This function turns off bounds checking by the animation system for all objects in the display list associated with handle. Turning off bounds checking will save some CPU time if your program can afford this luxury.

See also `gs_set_anim_bounds`

Function **gs_disable_anim_collision**
Disable an anim object from colliding with other objects

Synopsis

```
void gs_disable_anim_collision(anim);
                               a0

struct anim_struct *anim; /* pointer to anim structure */
```

Libraries none

Description This function disallows an active anim object and all attached anims (if any) from colliding with other active objects in the same display list. The object(s) may still be active and displayed through the animation system, even removed and readded at some point, but they will no longer collide with other objects in the system until re-enabled.

The anim object need not be active in the animation system before calling this function.

See also `gs_enable_anim_collision`

Function **gs_disable_anim_collision_bg**
Disable an anim object from colliding with background data

Synopsis

```
void gs_disable_anim_collision_bg(anim);
                               a0

struct anim_struct *anim; /* pointer to anim structure */
```


<i>Libraries</i>	none
<i>Description</i>	This function disallows an active anim object and all attached anims (if any) from colliding with background data. The object(s) may still be active and displayed through the animation system, even removed and re-added at some point, but they will no longer collide with background data until reenabled. The anim object need not be active in the animation system before calling this function.
<i>See also</i>	<code>gs_enable_anim_collision_bg</code>
<i>Function</i>	gs_disable_cplx_collision Disable an anim complex from colliding with other objects
<i>Synopsis</i>	<pre>void gs_disable_cplx_collision(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function disallows an active anim complex and all attached anims (if any) from colliding with other active objects in the same display list. The object(s) may still be active and displayed through the animation system, even removed and re-added at some point, but they will no longer collide with other objects in the system until re-enabled. The anim complex need not be active in the animation system before calling this function.
<i>See also</i>	<code>gs_enable_cplx_collision</code>
<i>Function</i>	gs_disable_cplx_collision_bg Disable an anim complex from colliding with background data
<i>Synopsis</i>	<pre>void gs_disable_cplx_collision_bg(anim); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function disallows an active anim complex and all attached anims (if any) from colliding with background data. The object(s) may still be active and displayed through the animation system, even removed and re-added at some point, but they will no longer collide with background data until reenabled. The anim complex need not be active in the animation system before calling this function.
<i>See also</i>	<code>gs_enable_cplx_collision_bg</code>
<i>Function</i>	gs_display Create a ViewPort display
<i>Synopsis</i>	<pre>display=gs_display(width, height, depth, modes, flags, color); struct display_struct *display; /* return display struct*/</pre>

```

int width; /* width of display in pixels */
int height; /* height of display in pixels */
int depth; /* depth of display (bitplanes) */
int modes; /* display mode bit flags */
int flags; /* display structure flags */
unsigned long *color; /* pointer to color table for display */

```

Libraries **GRAPHICS**

Description This is a much simpler form of the function `gs_create_display`. It does not require that you first set up a display and viewport structure, but in fact builds them for you and calls `gs_create_display`.

Note that this function ALWAYS allocates the necessary bitmap(s) for the display. If you want to supply your own, then you need to call `gs_create_display` with a properly filled display structure. In addition, if you require a scrollable display, you must use `gs_create_display`.

The color table must contain as many entries as are possible with the supplied depth. For instance, a 5 bitplane display can have up to 32 colors, and so the color table must contain 32 entries. Color table entries must be in standard GameSmith format.

Like `gs_create_display`, you must call `gs_show_display` before the new display is actually shown on the screen.

WARNING: This function takes over the Amiga's entire display. If you need Intuition compatibility, then use Intuition screens.

ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that all input parameters be placed on the stack.

Returns If successful, a pointer to a GameSmith display structure is returned.
Otherwise NULL is returned

Example

```

/*
Build a double buffered display and show access to the bitmap pointers.
*/

#define DEPTH 2
#define FLAGS GSV_DOUBLE

unsigned long color_table[1<<DEPTH] =
{
0x00ae8f43;      /* color 0 */
0x00341987;      /* color 1 */
0x00005600;      /* color 2 */
0x00ffffff;      /* color 3 */
};

int lores_display(int,unsigned long *);
int hires_display(int,unsigned long *);

main ( )
struct display_struct *d;
int dlist;

if (gs_open_libs(GRAPHICS,0))
    exit(01);

```

```

if(!(d=lores_display(DEPTH,color_table)))
{
    gs_close_libs();
    exit(02);
}
if ((dlist=gs_get_display_list()) < 0)
{
    gs_close_libs();
    exit(03);
}
gs_init_anim(dlist,d->vp->bitmap1,d->vp->bitmap2);
/* Your stuff here */

gs_free_display_list(dlist); /* release display list */
gs_remove_display(d); /* clean up our display*/
gs_close_libs(); /* close graphics library */
}
/***** /
int lores_display(depth,color)
int depth;
unsigned long *color;
{ return (gs_display(320,200,depth,0,FLAGS,color));
}
/ ***** /
int hires_display(depth,color)
int depth;
unsigned long *color;
{
return(gs_display(640,400,depth,HIRES|LACE,FLAGS,color));
}

```

See also `gs_create_display`, `gs_show_display`, `gs_flip_display`, `gs_old_display`, `gs_remove_display`

Function **gs_draw_anim_objs**
Copy all active anim objects to a bitmap

Synopsis

```

void gs_draw_anim_objs(handle);
                        d0

int handle; /* display list handle */

```

Libraries **GRAPHICS**

Description Sometimes you may need to do your own drawing to a bitmap, and you will want those results to appear behind anim objects in the bitmap. To accomplish this, you will need to use a `gs_restore_anim_bg/gs_draw_anim_objs` combination. Between these two function calls you will be able to do your own drawing (lines, block copies, or whatever). Note that the preceding function call to the animation system **MUST** be `gs_restore_anim_bg`, as this function relies on setup information by that function.

This function copies all active objects in a display list to the current drawing bitmap using the appropriate copy method for each object. In addition, this function handles collision detection.

A `gs_restore_anim_bg/gs_draw_anim_objs` combination has exactly the same effect as a single call to `gs_draw_anims`.

<i>Function</i>	gs_draw_anims Draw all active anims in a display list
<i>Synopsis</i>	<pre>void gs_draw_anims(handle); d0 int handle; /* display list handle */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This is the function that makes everything happen. It draws all active anim objects in the specified display list. It also takes care of restoring and saving background data, collision detection, object removal, and object invisibility.
<i>See also</i>	gs_init_anim, gs_restore_anim_bg, gs_draw_anim_objs, gs_get_display_list
<i>Function</i>	gs_enable_anim Make an active anim object visible again
<i>Synopsis</i>	<pre>void gs_enable_anim(anim); a0 struct anim_struct *anim; /* pointer to anim struct */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function makes an active anim object and all of its attached objects (if any) visible again. This reverses the effects of the <code>gs_clear_anim</code> function. The object(s) become visible again with the next call to <code>gs_draw_anims</code> . The object(s) will also be able to collide with other objects and background data again.
<i>See also</i>	gs_clear_anim
<i>Function</i>	gs_enable_anim_bounds Turn on bounds checking for objects in a display list
<i>Synopsis</i>	<pre>void gs_enable_anim_bounds(handle); d0 int handle; /* display list handle */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function turns on bounds checking by the animation system for all objects in the display list associated with handle. Bounds will be the same as last setting. If your program has not set any bounds, then the bounds are the entire 32K coordinate space.
<i>See also</i>	gs_set_anim_bounds, gs_disable_anim_bounds
<i>Function</i>	gs_enable_anim_collision Enable an anim object to collide with other objects
<i>Synopsis</i>	<pre>void gs_enable_anim_collision(anim); a0</pre>

```
struct anim struct *anim; /* pointer to anim structure */
```

Libraries none

Description This function allows an active anim object and all attached anims (if any) to collide with other active objects in the same display list. Anims created in CITAS that have object-to-object collision areas are automatically enabled.

If your program will not be supplying a collision handler function, it is only a waste of CPU time to allow the animation system to perform collision detection on active anims. The anim object need not be active in the animation system before calling this function.

See also `gs_disable_anim_collision`

Function **gs_enable_anim_collision_bg**
Enable an anim object to collide with background data

Synopsis

```
void gs_enable_anim_collision_bg(anim);
                                a0
```

```
struct anim struct *anim; /* pointer to anim structure */
```

Libraries none

Description This function allows an active anim object and all attached anims (if any) to collide with background data. Anims created in CITAS that have object-to-background collision points are automatically enabled.

If your program will not be supplying a collision handler function, it is only a waste of CPU time to allow the animation system to perform collision detection against background data.

The anim object need not be active in the animation system before calling this function.

See also `gs_disable_anim_collision_bg`

Function **gs_enable_cplx**
Make an active anim complex visible again

Synopsis

```
void gs_enable_cplx(cplx);
                   a0
```

```
struct anim cplx *cplx; /* pointer to anim complex */
```

Libraries none

Description This function makes an active anim complex and all of its attached anims (if any) visible again. This reverses the effects of the `gs_clear_cplx` function. The object(s) become visible again with the next call to `gs_draw_anims`. The object(s) will also be able to collide with other objects and background data again.

See also `gs_clear_cplx`

<i>Function</i>	gs_enable_cplx_collision Enable an anim complex to collide with other objects
<i>Synopsis</i>	<pre>void gs_enable_cplx_collision(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function allows an active anim complex and all attached anims (if any) to collide with other active objects in the same display list. Anims created in CITAS that have object-to-object collision areas are automatically enabled If your program will not be supplying a collision handler function, it is only a waste of CPU time to allow the animation system to perform collision detection on active anims. The anim complex need not be active in the animation system before calling this function.
<i>See also</i>	gs_disable_cplx_collision
<i>Function</i>	gs_enable_cplx_collision_bg Enable an anim complex to collide with background data Synopsis void
<i>Synopsis</i>	<pre>gs_enable_cplx_collision_bg(cplx); a0 struct anim cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function allows an active anim complex and all attached anims (if any) to collide with background data. Anims created in CITAS that have object-to-background collision points are automatically enabled. If your program will not be supplying a collision handler function, it is only a waste of CPU time to allow the animation system to perform collision detection against background data. The anim complex need not be active in the animation system before calling this function.
<i>See also</i>	gs_disable_cplx_collision_bg
<i>Function</i>	gs_file_requestor Get file and path information from the user
	<pre>status = gs_file_requestor(screen,title,file,hide,show, rows); int status; /* return status */ struct Screen *screen; /* pointer to screen on which to open the file requestor */ char *title; /* title to be placed in the file requestor's window border */ char *file; /* ptr to path & file display info as well as result string. */</pre>

```
char *hide; /* pointer to hide mask */
char *show; /* pointer to show mask */
int rows; /* number of rows to allocate for scrollable list window */
```

Libraries DOS, INTUITION, GRAPHICS

Description This function is compatible ONLY with the SAS/C development system, as it requires linkage with their SC.LIB linker library. This is the only function in the entire GameSmith Development System with this restriction, and is provided merely as an added convenience for SAS/C users.

The title string in this case would be "SelectIFF image to load". The file can either be a valid path/drawer (including optional device name) and file name string, or an empty string (the first character is a NULL). If the path portion of the file string is invalid, or the file string is empty, then the current directory is used. The deepest name of the input file string need not exist (i.e. if the user filled the file string with "work:gfx/pix/iff/testpic", the name "test.pic" need not exist in the subdirectory "iff"). If it does not exist, then it is assumed to be a file that might potentially be created, and it is placed in the 'File' input field of the requestor. In any case, the area pointed to by file must be large enough to hold the resultant path. AmigaDOS path names maybe up to 107 characters in length, so a string of 108 characters should be sufficient (one character for a terminating NULL).

The hide and show pointers are pointers to NULL terminated strings which contain masking information for the display window. The PC type wildcard characters may be used as part of the strings. For more information on the types of pattern matching available, see the function stepma in the SAS/C Library Reference manual. The hide string will restrict all matching files from the display list, while the show string is the master display filter; only strings matching that of the show string will be shown. Note that if the show string is an empty string upon calling this function it has the same effect as passing „*„. Also that directory names can not be masked from the list. The "Hide info" button is provided as an extra convenience when displaying a file list, and it is always the default. Clicking on this button will change it to read "Show info", and any "info" files in the current directory will show up in the list window.

The "Vols" button will discard the current directory name in the Drawer input field, and present the user with a list of AmigaDOS mounted volumes and devices. The "Asn" button is short for "assigns", and presents a list of assigned devices.

The "Parent" button moves back one subdirectory level, and presents a list of file and directory names in the parent directory. E.g. if the Drawer input field contained "work:gfx/pix/iff", then clicking on the "Parent" button would present a list for "work:gfx/pix".

Returns If the user clicked on the "OK" button, or hit enter in the 'File' input field, then the status return is equal to zero, and the file string is filled with the path that the user selected. Note that it is possible that the user entered an invalid file name at the end of the path, and your program should check for this situation.

The following are possible error returns:

- 1 Requestor window won't fit on the screen. More than likely your program tried to open the requestor with too many rows. The largest number of rows possible on a lores display is 10.

- 2 AmigaDOS could not open the requestor window. This is probably a low memory problem.
- 3 The user aborted the request by either clicking on the close window gadget or the "Cancel" button.

Function **gs_flip_display**
Flip to another ViewPort display

Synopsis `void gs_flip_display(display, sync);`

```
struct display_struct *display; /* display to flip */
int sync;      /* vertical blank sync value */
```

Libraries GRAPHICS

Description This function is used in a double buffered display environment to flip back and forth between two views previously set up by `gs_display` or `gs_create_display`. Double buffering is a simple technique which allows the programmer to draw into one view while displaying another. When the drawing is completed, you call this function to display the updated graphics in the other view. This provides the smoothest form of animated or moving graphics because the user does not see the screen updates taking place.

The hidden view is displayed by this function, and the other view is made hidden. This function is useless to a single buffered display, but is nonetheless safe to call in that environment.

It is up to the programmer to keep track of which view is currently being shown (the 1st or the 2nd). However, if all graphic objects are controlled through the GameSmith animation system, then you need not keep this information separately in your program. Always combine the animation function call `gs_next_anim_page` with this one (order is not important) and you won't have any problems.

The input parameter `sync` specifies the number of vertical blank periods that should happen before displaying the new view. Normally you will use a value of 1, which means to display the hidden view after the next vertical blank interrupt. This guarantees a rock solid display. If `sync` is zero, then the view is immediately loaded, which may cause display flicker.

ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the display and sync values (32 bit long words) be placed on the stack before making the call. Refer to the User's Guide for more information on calling 'C' functions from assembler.

See also `gs_display, gs_create_display, gs_show_display, gs_old_display,`
`gs_remove_display, gs_next_anim_page`

Function **gs_free_anim**
Free memory used by an anim object array

Synopsis `void gs_free_anim( anim, array_elements );`

```
struct anim_struct *anim; /* pointer to anim structure */
int array_elements;      /* number of elements in array */
```


<i>Libraries</i>	none
<i>Description</i>	This function is used to free up all resources for an array of simple anim objects previously loaded from disk via the <code>gs_load_anim</code> function. Make sure that you specify the same number of array elements that was used when loading the object. ASSEMBLER PROGRAMMERS: This function is written in ‘C’, and requires that the address of the anim structure and the number of array elements be placed on the stack before making the call.
<i>See also</i>	<code>gs_load_anim</code>
<i>Function</i>	gs_free_anim_save_area Free the background save area(s) for an anim object
<i>Synopsis</i>	<pre>void gs_free_anim_save_area(anim); a0 struct anim_struct *anim; /* pointer to anim structure */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function frees any background save areas for the specified anim object. Any attached anims are ignored. This function is always safe to call, since save areas will only be deallocated if they exist for the object. Always remember to call this function before exiting your program for anims that have the ANIM_SAVE_BG flag set. For attached anims, you may wish to call <code>gs_free_parent</code> instead.
<i>See also</i>	<code>gs_get_anim_save_area</code>
<i>Function</i>	gs_free_bitmap Free bitmap memory
<i>Synopsis</i>	<pre>void gs_free_bitmap(bitmap); struct BitMap *bitmap; /* pointer to bitmap structure */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function frees a bitmap previously allocated by a call to <code>gs_get_bitmap</code>. ASSEMBLER PROGRAMMERS: This function is written in ‘C’, and requires that the address of the bitmap structure be placed on the stack before making the call.
<i>See also</i>	<code>gs_get_bitmap</code>
<i>Function</i>	gs_free_blitter Deallocate use of the blitter
<i>Synopsis</i>	<pre>void gs_free_blitter(void);</pre>

<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function tells the system and the GameSmith blitter routines that other tasks may use the blitter. Only call this function after a call to <code>gs_get_blitter()</code> .
<i>See also</i>	<code>gs_get_blitter</code>
<i>Function</i>	gs_free_cplx Free memory used by a complex anim object array
<i>Synopsis</i>	<pre>void gs_free_cplx(cplx, array_elements); struct anim cplx *cplx; /* pointer to anim complex structure */ int array_elements; /* number of elements in array */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function is used to free up all resources for an array of complex anim objects previously loaded from disk via the <code>gs_load_anim</code> function. Make sure that you specify the same number of array elements that was used when loading the object. ASSEMBLER PROGRAMMERS: This function is written in ‘C’, and requires that the address of the complex anim structure and the number of array elements be placed on the stack before making the call.
<i>See also</i>	<code>gs_load_anim</code>
<i>Function</i>	gs_free_cplx_save_area Free the background save areas for an anim complex and all attached children
<i>Synopsis</i>	<pre>void gs_free_cplx_save_area(cplx); a0 struct anim_cplx *cplx; /* pointer to anim complex */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function frees any background save areas for the specified complex animation object and all attached anims (if any). This function is always safe to call, since save areas will only be deallocated if they exist for each object. Always remember to call this function before exiting your program for complex anims that have the ANIM_SAVE_BG flag set in the anim structures.
<i>See also</i>	<code>gs_get_cplx_save_area</code>
<i>Function</i>	gs_free_display_list Free memory associated with a display list
<i>Synopsis</i>	<pre>void gs_free_display_list(handle); d0 int handle; /* display list handle</pre>

<i>Libraries</i>	none
<i>Description</i>	This function releases all resources associated with a display list. The animation system no longer knows about the display list, and the handle is no longer valid.
<i>See also</i>	<code>gs_get_display_list</code>
<i>Function</i>	gs_free_hard_copper Free the list pointed by the LOF_cop and SHF_cop
<i>Synopsis</i>	<pre>void gs_free_hard_copper(hc); struct hard_copper *hc; /* pointer to hard copper structure */</pre>
<i>Description</i>	This function frees the list(s) pointed to by the LOF_cop and SHF_cop fields of the hard_copper structure. CAUTION!!! Because the copper may still be executing an old list after you call <code>gs_replace_ucop()</code>, you should wait an entire vertical blank period before freeing the old list(s).
<i>See also</i>	<code>gs_replace_ucop()</code>
<i>Function</i>	gs_free_parent_save_area Free the background save areas for a parent anim and all attached children
<i>Synopsis</i>	<pre>void gs_free_parent_save_area(anim); a0 struct anim_struct *anim; /* pointer to parent anim */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function frees any background save areas for the specified anim object and all attached anims (if any). This function is always safe to call, since save areas will only be deallocated if they exist for each object. Always remember to call this function before exiting your program for anims that have the ANIM_SAVE_BG flag set.
<i>See also</i>	<code>gs_get_parent_save_area</code>
<i>Function</i>	gs_free_sound Free sound data memory
<i>Synopsis</i>	<pre>void gs_free_sound(sound); a0 struct sound_struct *sound; /* sound struct with valid data pointer */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function frees the memory previously allocated by a successful call to either <code>gs_load_iff_sound</code> or <code>gs_load_raw_sound</code> .

See also `gs_load_iff_sound, gs_load_raw_sound`

Function **gs_free_spvp**
Free memory for a sliced parallax viewport

Synopsis

```
gs_alloc_spvp(spvp);

struct gs_spvp *spvp;    /* pointer to spvp structure */
```

Libraries none

Returns none

Function **gs_get_anim_save_area**
Allocate background save area(s) for an anim object

Synopsis

```
status = gs_get_anim_save_area(anim, flags);
      d0                      a0      d0

int status;                /* return status */
struct anim_struct *anim; /* pointer to anim structure */
int flags;                 /* allocation flags */
```

Libraries none

Description This function allocates one or two background save areas for the specified anim. Any attached anims are ignored. If the object is going to be used in a single buffered display, then only one save area is needed. In this case flags should be SINGLE BUFF. Otherwise two save areas are needed, and flags should be DOUBLE BUFF. No other flags are defined at this time.

If the anim has the ANIM CPUBLIT flag set, save areas are allocated from Fast RAM (if available). Otherwise save areas are allocated from Chip RAM, since the blitterchip will handle the copying of background data during the display of an anim object. For every attached anim, the ANIM CPUBLIT flag determines this same choice of RAM type. The save area(s) allocated will be big enough to hold the largest image in the animation sequence.

Note that save area(s) will only be allocated if the ANIIM_SAVE_BG flag is set in the anim structure.

NOTE: Save areas are automatically allocated if need be when an anim object is added to the animation system, so you generally won't need to call this function. However, if you should ever need to allocate a save area yourself, this function is available.

Returns If the save area(s) are allocated successfully, then a value of zero is returned. Other possible return codes are:

- 1 not enough Chip RAM
- 2 save area(s) already allocated

See also `gs_free_anim_save_area, gs_add_anim`

<i>Function</i>	gs_get_bitmap Allocate a bitmap
<i>Synopsis</i>	<pre> bitmap=gs_get_bitmap(depth,width,height,flags); struct BitMap *bitmap; /* pointer to bitmap structure */ int depth; /* tt of bitplanes to allocate */ int width; /* width of bitmap in pixels */ int height; /* bitmap height in scan lines */ int flags; /* 3.0 allocation flags */ </pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function provides an easy way to allocate a bitmap structure and the necessary bitplanes that form a display. In addition, the bitplanes are cleared (filled with zero), and the plane size is guaranteed to be aligned properly so that it is displayable on any machine in any display mode. The flags parameter applies only to machines running Workbench 3.0 or later. You will usually never pass any flag values other than 0. ASSEMBLER PROGRAMMERS: This function is written in ‘C’, and requires that all input parameters be placed on the stack before making the call.
<i>Returns</i>	If successful, a pointer to the allocated bitmap structure is returned. Otherwise NULL is returned, indicating there was not enough RAM available.
<i>See also</i>	gs_free_bitmap
<i>Function</i>	gs_get_blitter Allocate use of the blitter
<i>Synopsis</i>	<pre> void gs_get_blitter(void); </pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function merely calls the system function OwnBlitter(), but with one addition: the GameSmith blitter routines know about the call. In general, you will probably never need to use this function unless you intend to make quite a number of calls to the GameSmith blitter routines. You don’t even have to use this function, the GameSmith blitter routines are smart enough to allocate the blitter, but it may save a few processing cycles if you need to make a lot of blitter routine calls. If using only the animation system, never call this function. Remember to call <code>gs_free_blitter</code> when you’re finished making the blitter calls so that other tasks may use the blitter.
<i>See also</i>	gs_free_blitter
<i>Function</i>	gs_get_cplx_save_area Allocate background save areas for an anion complex and all attached children
<i>Synopsis</i>	<pre> status = gs_get_cplx_save_area(cplx, flags); a0 d0 int status; /* return status */ </pre>

```
struct anim cplx *cplx; /* pointer to anim complex */
int flags;             /* allocation flags */
```

Libraries none

Description This function allocates one or two background save areas for the specified anim complex and all attached anims. If the object is going to be used in a single buffered display, then only one save area is needed. In this case flags should be SINGLE_BUFF. Otherwise two save areas are needed, and flags should be DOUBLE_BUFF. No other flags are defined at this time.

One of the great advantages of a complex animation object is that it only requires one or two save areas for the entire object, allowing all animation sequences within the object to use the same save area(s). The save area(s) allocated will be big enough to handle the largest image in the entire complex (the largest image from all anim objects within the complex). It is possible for one or more of the anim objects within the complex to have attached anims. Attached anims do NOT use the same save areas, but are allocated save areas of their own. When calling this function, none of the anim objects within the complex should have save areas already allocated or those areas will be lost and forgotten (except save areas for attached anims).

If the first anim within the complex has the ANIM_CPUBLIT flag set, save areas are allocated from Fast RAM (if available). Otherwise save areas are allocated from Chip RAM, since the blitter chip will handle the copying of background data during the display of an anim object. For every attached anim, the ANIM_CPUBLIT flag determines this same choice of RAM type.

IMPORTANT: Save areas will only be allocated if the ANIM_SAVE_BG flag is set in the anim structure of the first anim object of the complex. If you want background data to be saved by the animation system, then make sure ALL anim objects within a complex have the ANIM_SAVE_BG flag set, or problems may arise.

NOTE: Save areas are automatically allocated if need be when an anim object is added to a display list, so you generally won't need to call this function. However, if you should ever need to allocate a save area yourself, this function is available.

Returns If all save areas are allocated successfully, then a value of zero is returned. Other possible return codes are:

- 1 not enough Chip RAM
- 2 one or more objects already had a save area allocated

Unlike `gs_get_parent_save_area`, in the event of a nonzero return, any and all save areas for objects within the complex (including attached anims) are deallocated. In the event of code 2, calling this function again should result in a successful completion. Also note that unlike `gs_get_parent_save_area`, code 2 IS a real error, and any save areas were released.

See also `gs_free_cplx_save_area`, `gs_add_anim cplx`

Function **`gs_get_display_list`**
Allocate a display list for use with the animation system.

<i>Synopsis</i>	<pre> handle = gs_get_display_list(void); d0 int handle; /* display list handle */ </pre>
<i>Libraries</i>	none
<i>Description</i>	The animation system uses "display lists" to keep track of what needs to be drawn and where. Associated with a display list are things like bitmap pointers for drawing, object boundaries, real space offsets, etc. You tell the animation system that you want an object drawn in a particular bitmap by adding it to an initialized display list. The animation system keeps a linked list of object for every display list that you allocate. This function allocates such a display list. Much like a file handle when you open a disk file, the animation system will return a display list handle associated with a display list. Many of the animation functions will require this handle as the first input parameter. WARNING: The animation system never checks the validity of a display list handle, so make sure you save the handle value in a safe place. All animation functions that require the handle as an input parameter assume that a proper display list will be associated with the handle. Default display list values: • Real space offset of 0,0 • Bounds checking ON • Object bounds of 0,0 to 32767,32767
<i>Returns</i>	If successful, a non-negative list handle is returned.
<i>See also</i>	<code>gs_free_display_list</code> , <code>gs_init_anim</code>
<i>Function</i>	gs_get_ILBM_bm Read header information from an IFF ILBM image file
<i>Synopsis</i>	<pre> status = gs_get_ILBM_bm(load) ; int status; /* return */ struct loadILBM_struct *load; /* pointer to load structure */ </pre>
<i>Libraries</i>	GRAPHICS, DOS
<i>Description</i>	This function does not actually attempt to load the image from disk, but merely reads header information from the file. The bitmap(s) fields <code>BytesPerRow</code>, <code>Rows</code>, and <code>Depth</code> are filled accordingly, as well as the other informational fields within the <code>loadILBM_struct</code> structure itself. If the <code>bmh</code> field is not NULL, then it is assumed to point to a valid <code>BitMapHeader</code> structure which is also filled. This function pays attention to the allocation flags, and first allocates 1 or 2 bitmaps if so specified. Otherwise, the load structure should contain valid pointers to at least one bitmap structure. If you wish to merely fill a <code>BitMapHeader</code> structure, you may call this function with both bitmap pointers equal to NULL. The function will return a -1 value, but the <code>BitMapHeader</code> structure will be filled.

This function calls `gs_get_bitmap` when allocating bitmaps. Your program must call `gs_free_bitmap` for each bitmap that was allocated before exiting.

See the GameSmith User's Guide for complete information on the `loadILBM_struct` structure and its fields.

This function is useful in determining the size and attributes of an image file before actually loading it, in determining if the file exists, or in determining if the file is a valid IFF ILBM type.

ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the address of the load structure be placed on the stack before making the call.

Returns If successful, zero is returned. Otherwise a nonzero value is returned. Possible error codes are:

- 1 not enough memory for bitmap(s) or no bitmap to fill
- 2 error opening file
- 3 error while reading file
- 4 unrecognized compression technique
- 5 unexpected end of file reached
- 6 not an IFF ILBM file
- 7 no BODY chunk (no image data)
- 8 no BMHD chunk (no header information)
- 9 unrecognized encryption algorithm

See also `gs_loadILBM`

Function **gs_get_parent_save_area**
Allocate background save areas for parent anim and all attached children

Synopsis `status = gs_get_parent_save_area(anim, flags);`
`d0                                   a0   d0`

```
int status;                           /* return status */
struct anim_struct *anim; /* pointer to parent anim */
int flags;                            /* allocation flags */
```

Libraries none

Description This function allocates one or two background save areas for the specified anim and all attached anims. If the object is going to be used in a single buffered display, then only one save area is needed. In this case flags should be SINGLE BUFF. Otherwise two save areas are needed, and flags should be DOUBLE BUFF. No other flags are defined at this time.

If the anim has the ANIM CPUBLIT flag set, save areas are allocated from Fast RAM (if available). Otherwise save areas are allocated from Chip RAM, since the blitter chip will handle the copying of background data during the display of an anim object. For every attached anim, the ANIM CPUBLIT flag determines this same choice of RAM type. Each save area allocated will be big enough to hold the largest image in each object.

Note that save areas will only be allocated if the ANIM_SAVE_BG flag is set in the anim structure.

NOTE: Save areas are also allocated if need be when adding an anim object to a display list. However, some situations may require that background save areas be allocated before an object is added to the animation system. In general, you will probably never need to call this function.

Returns If all save areas are allocated successfully, then a value of zero is returned. Other possible return codes are:

- 1 not enough Chip RAM
- 2 one or more objects already have a save area allocated

Note that a return code of 2 is not really an error. All save areas were successfully allocated. This is to inform you that your program tried to allocate a save area twice for one or more objects.

In the event that code 1 is returned, any save areas in the object chain are NOT released upon return. To make sure all save areas are freed, call `gs_free_parent_save_area`.

See also `gs_free_parent_save_area`, `gs_add_parent`

Function **gs_hard_copper**
Build a hardware list out of a user copper list

Synopsis

```
status = gs_hard_copper(hc, struct gs_viewport *);

int status; /* return status */
struct hard_copper *hc; /* pointer to hard_copper structure */
struct gs_viewport *vp; /* pointer to target viewport which will become
the parent of the user copper list */
```

Description This function builds a hardware list or lists out of a user copper list array pointed to within the `hard_copper` structure (see below).

Returns

- 0 Everything went OK, list(s) are valid
- 1 Insufficient Chip RAM to contain list(s)
- 2 Illegal/unknown instruction encountered in mnemonic list

Upon successful return, the first `LOF_cop` pointer in the `hard_copper` structure will always be valid (point to an actual hardware list). Depending on the display mode (interlace) and double buffering, the other pointers maybe valid as well. If the pointer is NOT NULL, then it is valid, and points to a hardware user copper list.

For a good example of direct modification of hardware user copper lists for special effects, see the demo "bubble_copper" included on the example disks. Pay special attention to the manual section on copper instruction lengths.

Function **gs_init_anim**
Initialize the animation system

Synopsis

```
void gs_init_anim(handle ,bitmap1,bitmap2);
                   d0      a0      a1
```

`int handle; /* display list handle */`

```
struct BitMap *bitmap1,*bitmap2;
```

Libraries none

Description This function initializes a display list for use with the specified bitmap(s). This tells the animation system where to draw anim objects. If you want a double buffered display, then you must supply two bitmaps. If you don't want a double buffered display, pass NULL in place of the 2nd bitmap pointer. After calling this function to set up a double buffered display, the 2nd bitmap will be used first for drawing anim objects. This assumes that you will be displaying bitmap 1 first, and then switching to bitmap 2 after the first drawing phase. Also, the animation lists are cleared so that no objects exist in the display system, and all pointers to collision routines are cleared.

A call to this function **MUST** precede all other animation functions that reference the same display list within your program. Rarely will you ever need to initialize a display list more than once, but if you find it necessary, take care to remove all objects in the display list first.

See also `gs_next_anim_page`, `gs_flip_display`, `gs_draw_anims`, `gs_get_display_list`

Function **gs_init_vector**
Initialize a vector for use

Synopsis void
`gs_init_vector(vector, start_x, start_y, end_x, end_y);`
 d2 a0 a1 d0 d1

```
int vector; /* vector number (0 - 99) */
int start_x; /* starting X coordinate */
int start_y; /* starting Y coordinate */
int end_x; /* ending x coordinate */
int end_y; /* ending Y coordinate */
```

Libraries none

Description This function initializes one of the 100 built in vectors. A vector, by definition, is a line segment which has direction. That line segment is defined within the GameSmith system as having a beginning and ending point in 2D space, and given direction by specifying which end point is the start, and which is the end. The start and end of the vector are specified using common Cartesian X and Y coordinates. Movement along the vector is generated by calling either `gs_step_vector` or `gs_ustep_vector` to generate a new set of X,Y coordinates. The X,Y coordinates along the line are generally related exactly to bitmap X and Y offsets in order to control movement of a graphic object or to draw a line.

See also `gs_step_vector`, `gs_ustep_vector`

Function **gs_joystick**
Get joystick values for the specified port

Synopsis `stick = gs_joystick(port);`
 d0 d0

```
unsigned char stick; /* return bit value assignments */
int port;          /* port number to poll */
```

<i>Libraries</i>	none
<i>Description</i>	This function gets the current status of the joystick port specified.
<i>Returns</i>	An 8 bit unsigned character value is returned containing bit values for JOY_RIGHT, JOY_LEFT, JOY_DOWN, JOY_UP, JOY_BUTTON1, and JOY_BUTTON2. The bit values are TRUE (bit is on) if the joystick is pushed in a direction, or the joystick button is pressed.
<i>Example</i>	<pre> /* Poll joy port 1, and end if mouse (or joystick in port 0) button is pressed. Note that this could just as well poll for a joystick at port address 0. Note also the bitwise AND operation required to determine if the proper bit is set in the returned value. This is opposed to the boolean AND operation which determines if two conditions are true. */ main() { unsigned char stick; while (!(gs_joystick(0) & JOY_BUTTON1)) { /* loop until left mouse button pressed */ stick = gs_joystick(1); /* poll port 1 */ if(stick & JOY_DOWN) && /* if down and button */ (stick & JOY_BUTTON1) { megablast(); /* call megablast routine */ } } } </pre>
<i>Function</i>	gs_LEDoff Turn off the audio filter (LED dim)
<i>Synopsis</i>	<code>void gs_LEDoff(void);</code>
<i>Libraries</i>	none
<i>Description</i>	This function turns the Amiga's hardware audio filter OFF, and makes the LED on the front panel dim.
<i>See also</i>	gs_LEDOn
<i>Function</i>	gs_LEDOn Turn off the audio filter (LED bright)
<i>Synopsis</i>	<code>void gs_LEDOn(void);</code>
<i>Libraries</i>	none
<i>Description</i>	This function turns the Amiga's hardware audio filter ON, and makes the LED on the front panel bright.
<i>See also</i>	gs_LEDoff

<i>Function</i>	gs_load_anim Load an animated graphic object from disk
<i>Synopsis</i>	<pre>status = gs_load_anim(load); int status; /* return */ struct anim load_struct *load; /* pointer to load structure */</pre>
<i>Libraries</i>	DOS
<i>Description</i>	This function loads an animated graphic object into memory from a disk file. The graphic object must have been previously created using CITAS. The result is a pointer to an array of anim objects, with the number of elements in the array being equal to the array_elements field of the load structure. The type of the object will be filled at load time. At this time, two object types exist simple anim objects (single animation sequence), and complex anim objects (multiple animation sequences). This function loads both types impartially. Note, however, that there are separate functions for freeing the two different types of anim objects. Once the anim object(s) are loaded using this function, they are directly usable by the animation system with no additional attention required by your program. You need only fill the filename, cmap_size, array_elements, and flags fields of the load structure. The remaining fields are initialized and filled by the load function. Since CITAS allows the option to use either the Amiga's blitter chip or the CPU for image display, the actual graphic data is loaded to either Chip RAM or Fast RAM depending on this option setting. If the object is CPU-blittable, then the graphic data will be loaded to Fast RAM if available. Your program may also opt to override object settings by using either the ANIMLOAD_FASTRAM or ANIIMLOAD_NOFASTRAM flag bits at load time. NOTE: If you allow this function to allocate a color table for the object, be sure to free the memory used for the table before exiting your program. Use the Exec function FreeMem to free the table. Each entry in the table is a 4 byte long integer. This function allocates the complex framework of structures and data that make up a single graphics object, whatever its type may be. This allows your program to treat graphics objects as just that, an object (via the pointer returned in the load structure), which is easily placed and moved on the screen. Most of the time your program doesn't even need to know the size, shape, color of the object, how many frames make up a particular animation sequence, or how many and where collision points are in the object. The object itself contains all of this information, adding a degree of inherent "intelligence" to your graphics. Extrapolating this process leads to all sorts of exciting possibilities in interchangeable and modifiable objects used by a single program. ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the address of the load structure be placed on the stack before making the call.
<i>Returns</i>	If successful, zero is returned. Otherwise a nonzero value is returned. Possible error codes are: -1 couldn't open file. File probably doesn't exist -2 error reading file

- 3 not a recognized anim file
- 4 error in file format
- 5 unexpected end of file
- 6 insufficient memory to load anim
- 7 user load parameter error. Check load structure setup
- 8 file is locked, and user does not have proper key
- 9 if ANIMLOAD_CHECK is set and the anim object is not locked

Example

```

/*
The following code fragment shows an example of loading a complex anim
object and adding it to the animation system. The object is used as a
basis to determine the depth and color of the display. This particular
object is a frog animation with 2 sequences; one jumping left and one
jumping right. You could add another animation sequence of the frog
eating a fly, for instance.
This example might be typical of a setup function within your program. */
#include "frog.h"
#define SWIDTH 320 /* screen width */
#define SHEIGHT 200 /* screen height */
#define FROG_CNT 5 /*number of frogs to alloc */
#define FROG_INIT Y 180 /* initial frog Y placement

int err,depth,x[FROG_CNT],direction[FROG_CNT];
struct anim_cplx *frog;
struct anim_struct *anim;
struct blit_struct *img;
struct display_struct *display;

struct anim_load_struct load =
{
    "frog", /* name of anim file to load */
    NULL, /* ptr to anim upon return */
    NULL, /* ptr to attachment array specification upon return */
    NULL, /* ptr to color map upon return */
    0, /* number of entries in the color map upon return */
    8, /* number of bits per color map entry */
    0, /* type (filled). 0 = anim, 1 = complex */
    FROG_CNT, /* number of frogs to allocate in array */
    0 /* flags (alloc color map) */
};

if (err=gs_load_anim(&load)) /* load 5 frog objects */
return(err) ;
if (!load.type) /* make sure it's a complex anim */
{ /* if not, free everything & abort */
FreeMem(load.cmap,load.cmap_entries*sizeof(long));
gs_free_anim(load.anim,load.array_elements);
return(-99);
}
frog = load.anim_ptr.cplx; /* pointer to 1st anim complex */
anim = frog[0].list; /* pointer to 1st anim complex */
while (anim)
{
img = anim->list; /* pointer to 1st image in anim sequence */
while (img) /* find max depth of anim */
{
if (img->depth > depth)
depth = img->depth;
if (img->next == img) /* avoid infinite loop */
img=NULL;
else
img=img->next;
}
}
}

```

```

    }
    anim=anim->cplx_next; /* next anim in complex */ }
    }
    if
    (! (display=gs_display(SWIDTH,SHEIGHT,depth, 0,GSV_DOUBLE, load.cmap))) /*
    allocate double buffered display*/
    { /* if display setup error, free everything & abort */
    FreeMem(load.cmap, load.cmap_entries*sizeof(long));
    gs_free_cplx(frog,FROG_CNT);
    return(-98) ;
    }
    /* free color map. We don't need it any more */
    FreeMem(load.cmap,load.cmap_entries*sizeof(long));
    /* allocate a display list for anims */
    if ((dl=gs_get_display_list()) < 0)
    {
    gs_free_cplx(frog,FROG_CNT);
    return(-97) ;
    } gs_init_anim(dl,display->vp->bitmap1, display->bp->bitmap2);
    gs_set_anim_bounds(dl,0,0,SWIDTH-1,SHEIGHT-1);
    for (cnt=0; cnt < FROG_CNT; cnt++)
    {
    x[cnt] = gs_random(SWIDTH); /* random X coordinate*/
    if (gs_random(2)) /* pick random direction for frog*/
    direction[cnt]=FROG_LEFT;
    else
    direction[cnt]=FROG_RIGHT;
    if (err=gs_add_anim_cplx(dl,&frog[cnt],x[cnt],FROG_INIT Y,
    direction[cnt],frog[cnt].list->prio))
    {
    gs_free_cplx(frog,FROG_CNT);g
    gs_remove_display(display);
    return(err); /* return failure */
    }
    /* pick random image from current anim sequence */
    gs_set_cplx_cell(&frog[cnt], gs_random(frog[cnt].anim->count);
    }
    gs_draw_anims(dl); /* draw into 2nd bitmap */
    gs_next_anim_page(dl); /* use next bitmap (1st) */
    gs_flip_display(display,1); /* show results of drawing */
    return(0);
    )

```

See also gs_free_anim, gs_free_cplx

Function **gs_load_iff_sound**
Load an IFF 8SVX sound sample from disk

Synopsis

```

status = gs_load_iff_sound(sound1,sound2,file);
d0          a0          d0          a1

int status /* return */
struct sound_struct *sound1 /* sound struct to fill*/
struct sound_struct *sound2 /* 2nd struct for stereo */
char *file; /* ASCII name of file to load */

```

Libraries DOS, GRAPHICS

Description This function loads an IFF8SVX sound sample into memory from a disk file. The file name must be in null terminated ASCII format.

There are two ways in which the load routine may allocate memory to hold the file data. If the SND_FAST flag is set in the sound structure, then memory is allocated from Fast RAM if available. If not enough Fast RAM is available, Chip RAM is used. If the SND_FAST flag is not set, then memory is allocated directly from Chip RAM.

This function has the ability to load stereo sound samples. Stereo sound sample files contain two separate sound samples designed to play simultaneously; one through a left channel, and one through a right channel. In order to load the second sample, you must supply a pointer to a second sound structure. If the file being loaded is a stereo file, then the data pointer of the second structure will point to valid data. Otherwise the data pointer of the second structure will be NULL. You may opt to only load the fast sample from a stereo file by passing NULL in place of the second sound structure pointer.

In addition to stereo samples, 8SVX files may contain information about how a sample should repeat. This is useful for echo effects. If the file has repeat information, the loop field of the sound structure will point to the repeat portion of the sample instead of pointing to the start of the sample as is normal. Only the loop portion of the sample will be played after the first time through if the repeat field of the sound structure is greater than 1 when the sample is played.

One of the great advantages of the 8SVX format versus a raw sound sample is that it contains information about volume and playback frequency. This guarantees that your program can play back sound samples exactly the way they were recorded. Also, sample period values are automatically adjusted for differences in NTSC and PAL systems so that the sample will sound the same on either machine.

Returns If successful, zero is returned. In this case, the type field will always be set to SND_EFX, the repeat field is set to 1, and volfade, volcnt, perfade, and percnt are cleared to zero. At this point, the sound data is ready to be played through one or more sound channels.

Possible error returns are:

- 1 file not found, or unable to access
- 2 error reading file
- 3 insufficient memory for 1st sound sample
- 4 not an IFF 8SVX sound file
- 5 8SVX header incompatible size
- 6 missing 8SVX header
- 7 not enough memory for 2nd sample (stereo), but 1st sample loaded OK
- 8 unrecognized encryption algorithm

See also `gs_load_raw_sound`, `gs_start_sound`, `gs_free_sound`

Function **gs_load_raw_sound**
Load a raw sound sample from disk

Synopsis

```
status = gs_load_raw_sound(sound,file);
d0                                a0    al

int status /* return */
struct sound_struct *sound; /* sound struct to fill */
char *file; /* ptr to ASCII name of file to load */
```

<i>Libraries</i>	DOS
<i>Description</i>	This function loads a file into memory as unformatted sound data. The file name must be in null terminated ASCII format. There are two ways in which the load routine may allocate memory to hold the file data. If the SND_FAST flag is set in the sound structure, then memory is allocated from Fast RAM if available. If not enough Fast RAM is available, Chip RAM is used. If the SND_FAST flag is not set, then memory is allocated directly from Chip RAM.
<i>Returns</i>	If successful, zero is returned. In this case, all fields of the sound structure are initialized except flags. The loop pointer will be identical to the data pointer since an unformatted file has no loop portion. The repeat field is set to 1, period is set to 428 (C2 note value), volume is set to maximum (64), type is SND_EFX, and volfade, volcnt, perfade, and percnt are all cleared to zero. At this point, the sound data is ready to be played through one or more sound channels. Possible error returns are: 1 file not found 2 unable to access file 3 insufficient memory to hold file
<i>See also</i>	gs_load_iff_sound, gs_start_sound, gs_free_sound
<i>Function</i>	gs_LoadILBM Load an IFF ILBM image from disk
<i>Synopsis</i>	<pre>status = gs_loadILBM(load); int status; struct loadILBM struct *load; /*return pointer to load structure */</pre>
<i>Libraries</i>	GRAPHICS, DOS
<i>Description</i>	This function loads an IFFILBM file from disk and places it into one or two supplied bitmaps; optionally allocating the bitmap memory in the process. Additionally, the programmer may opt to have this function fill a color table with color values for the image. If this function does not allocate the bitmap(s) and the ILBM picture is larger than the supplied bitmap dimensions, then the picture is dipped to fill only the bitmap dimensions. Numerous fields of the loadILBM_struct structure are filled with information about the image after it has been loaded. See the GameSmith User's Guide for complete information on the loadILBM struct structure and its fields. Note that this function calls gs_get_bitmap when allocating bitmaps. In this event, your program must call gs_free_bitmap before exiting. ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the address of the load structure be placed on the stack before making the call.
<i>Returns</i>	Zero is returned if successful. Otherwise, possible error codes are:

- 1 not enough memory for bitmap(s)
- 2 error opening file
- 3 error while reading file
- 4 unrecognized compression technique
- 5 unexpected end of file readied
- 6 not an IFF ILBM file
- 7 no BODY chunk (no image data)
- 8 no BMHD chunk (no header information)
- 9 unrecognized encryption algorithm

See also `gs_get_ILBM_bm`, `gs_free_bitmap`

Function **gs_LoadRGB**
Reload the entire color palette of GameSmith viewport

Synopsis

```
status = gs_LoadRGB(display, vp, table);

int status;
struct display_struct *display; /* pointer to display structure */
int vp; /* viewport number to modify */
unsigned long *table; /* pointer to new color table */
```

Libraries GRAPHICS

Description This function reloads the entire color palette of a GameSmith viewport. The viewport number is a value from 0 (first and topmost in the display) to n (last and bottommost in the display). The table entries must be in standard GameSmith format, and the table itself must contain enough entries to cover the depth of the viewport.

ASSEMBLER PROGRAMMERS: This function was written in ‘C’, and therefore requires that all input parameters be passed on the stack. See the User’s Guide for more information about calling ‘C’ functions from assembler.

Returns If the viewport number is found, then the color palette is modified. Otherwise the value GSVP_NOVP is returned.

See also `gs_create_display`, `gs_SetRGB`

Function **gs_move_anim**
Reposition an anim object

Synopsis

```
void gs_move_anim(anim, x, Y);
                  a0    d0 d1

struct anim_struct *anim; /* pointer to anim struct */
int x,y; /* X,Y coordinates for object placement */
```

Libraries none

Description This function repositions an anim object at the specified x,y coordinates within virtual space. At this time, valid virtual space coordinates range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. The object is not animated (the cell number is not changed) by this function. Movement of a parent anim also moves all attached child anims.

X and Y coordinates refer to the upper left hand corner of an image.

See also gs_anim_obj, gs_draw_anims, gs_set_anim_bounds

Function **gs_move_cplx**
Reposition an anim complex

Synopsis

```
void gs_move_cplx(cplx, x, y);
                a0  d0 d1

struct anim cplx *cplx; /* pointer to anim complex */
int x,y;                /* X,Y coordinates for object placement */
```

Libraries none

Description This function repositions an anim complex at the specified x,y coordinates within virtual space. Valid virtual space values range from 0 to 32767 (0x7fff hex). Note that no actual drawing takes place as a result of this call. The object is not animated (the cell number is not changed) by this function. Movement of a parent complex also moves all attached child anims.

X and Y coordinates refer to the upper left hand corner of an image.

See also gs_anim_cplx, gs_draw_anims, gs_set_anim_bounds

Function **gs_move_sprite**
Moves a sprite to specified position

Synopsis

```
void gs_move_sprite(data,height,x,y);

unsigned short *data; /* pointer to actual sprite data bank */
int height; /* sprite height in scan lines
int x,y; /* new sprite X& Y coordinates
```

Libraries none

Description This function moves a sprite to the new X & Y screen coordinates (relative to the sprite's upper left hand corner). You can reposition a sprite at any time.

Returns none

Function **gs_next_anim_page**
Tell the animation system to use the other bitmap for drawing

Synopsis

```
page = gs_next_anim_page(handle);
d0                                d0

int page; /* new drawing page number (bitmap 0 or 1) */
int handle; /* display list handle */
```

Libraries none

Description This function instructs the animation system to use the next bitmap in a double buffered display for the next drawing phase. If the current page is 0 (1st bitmap), then page 1 (2nd bitmap) will be used during the next call to gs_draw_anims, or vice versa. This function

has no effect in a single buffered environment.

If this function is always paired with a call to `gs_flip_display` (order is not important), then you may choose to ignore the page number returned from this function.

Returns The new current drawing page is returned. This will be either page 0 or page 1, corresponding accordingly to the 1st or 2nd bitmap used by the animation system.

See also `gs_flip_display`, `gs_draw_anims`, `gs_init_anim`

Function **gs_old_display**
Show the previous system view display

Synopsis

```
void gs_old_display(display);  
  
struct display_struct *display;
```

Libraries GRAPHICS

Description This function calls the systems graphics function LoadView to display the Amiga view previously displayed before calling `gs_display` or `gs_create_display` to set up a new one. The display structure holds a pointer to the view that was previously displayed. Note that this call is only valid AFTER having called `gs_display` or `gs_create_display`. The previous view is normally the Intuition environment where screens and windows reside.

This is useful, for instance, if the programmer wants the ability to switch back to the Workbench screen.

See also `gs_display`, `gs_create_display`, `gs_show_display`, `gs_remove_display`

Function **gs_open_libs**
Open AmigaDOS libraries

Synopsis

```
status = gs_open_libs(libs,version);  
      d0          d0    d1  
  
int status; /* return status */  
int libs;   /* bit mask of the libraries to open */  
int version; /* version of the libraries to open*/
```

Libraries none

Description This function opens the user specified libraries, and retains a knowledge of which libraries were opened. For a list of libraries and their definitions, see the GameSmith.h header file or GameSmith.i include file. For a list of library version numbers, see Commodore's AMIGA ROM Kernel Reference Manual: Libraries. Specifying a library version of 0 will open the latest version available.

It is perfectly OK to call this function multiple times with the same library specifications. This makes it possible for different program modules to guarantee that the libraries they need are opened. However, only the first `open_libs` call to any given library actually opens the library.

To open multiple libraries with the same call, simply use a bitwise OR to combine the libraries you would like to open.

Returns If successful, a status of 0 will be returned. A non-zero value indicates that a library could not be opened. In the event of an error, ALL AmigaDOS libraries that had been opened by the `gs_open_libs` function will be closed prior to return of the function. Also, in the event of an error, the nonzero value returned to your program is actually a pointer to a NULL terminated string containing the name of the library that could not be opened. However, if not even DOS can be opened, it will be difficult for your program to indicate which library is causing a problem.

Example

```
/*
Open some AmigaDOS libraries
*/
main()
{
int lib_err;
if
(lib_err=gs_open_libs(DOS|INTUITION|GRAPHICS, 0))
{
/* if can't open a lib */
if (!gs_open_libs(DOS,0)) /* if able to open DOS */
{ /* let's see problem lib */
printf("can't open os\n", (char *)lib_err);
gs_close_libs(); /* close DOS */
exit(01); /* kill program */
}
else /* if can't even open DOS */
exit(01); /* kill program */
}
/* do the program */
gs_close_libs(); /* close all Amiga libs */
}
```

See also `gs_close_libs`

Function **gs_open_sound**
Open the interrupt driven sound system

Synopsis

```
status= gs_open_sound(alloc,filter,prio,buffsize);
d0          d0      d1      d2      d3

int status; /* return status */
int alloc; /* allocate sound channels flag */
int filter; /* low pass filter flag */
int prio; /* interrupt code priority */
int buffsize; /* number of words for playback
buffer */
```

Libraries none

Description This function opens the interrupt driven sound system, and prepares it for use by your program. The GameSmith sound system works directly with the sound DMA hardware, thus gaining the utmost in sound performance; yet it remains compatible with all Amiga systems.

The alloc flag, if nonzero, tells the sound system to allocate the audio.device and all sound channels for exclusive use. If the alloc flag is zero, then the hardware sound channels will be used directly without first telling the system that your program will be using them. This is not a very system friendly method, and can lead to system crash if other programs try to

use the sound channels (it may confuse other programs and cause them to crash).

The filter flag, if nonzero, tells the sound system to turn on the low pass filter (the system LED will be bright).

The input parameter prio allows your program to specify the priority level at which the vertical blank driver will operate. Valid priority levels range from -128 to 127, with 127 being the highest. Normally, vertical blank interrupt routines run at priority 0. A little experimentation may be necessary to determine the best use with your program. Note that the SOFTWARE interrupt caching system always operates at the highest possible level; this being 32.

The input parameter buffsize specifies the number of words (16 bit integers) to be allocated per sound buffer in Chip RAM for playing sound data. A total of eight separate buffers are allocated; two for each channel. Thus if you specify a buffer size of 2K words (4096 bytes), then a total of 32K bytes would be.

Note that buffer size is directly related to the amount of CPU time required to copy data to the buffer as needed. The larger the buffer, the less CPU expenditure. Also, the faster the data is played, the more often the sound data will need to be copied to the sound buffers. All of this operates transparently behind the scenes, but it is useful for the programmer to know in order to fine tune his/her programs. Oregon Research has determined that buffers as small as 256 words (512 bytes) work well on any Amiga; even during a graphics intensive display. However, a slightly larger buffer size is recommended if available.

Keep in mind that the sound buffers are allocated only one time when you open the sound system, and are used for all sound samples you wish to play through the system. Buffers are not allocated for every sound sample.

Additionally, you may opt NOT to allocate Chip RAM buffers. If your program is running on a system that does not have Fast RAM, then there is no need to allocate additional buffers because all sound samples need to be loaded directly to Chip RAM anyway.

Returns If successful, zero is returned. Otherwise a nonzero value is returned. Generally, if this function fails it is because it was unable to allocate the audio.device for exclusive use, or it was unable to allocate the necessary Chip RAM buffers.

See also `gs_close_sound`, `gs_start_sound`, `gs_stop_sound`, `gs_load_raw_sound`, `gs_load_iff_sound`

Function **gs_open_vb_timer**
Open the custom vertical blank timer

Synopsis `void gs_open_vb_timer(void);`

Libraries none

Description This function installs a timer routine into the system's vertical blank interrupt server chain so that a counter is incremented every vertical blank interval (60 times a second for NTSC displays, and 50 times a second for PAL displays). The user may poll this counter for timing information. The timer routine itself is all of 3 machine instructions, providing a very fast, efficient timing method.

Note that a program should only call this function once. This function does not install a new timer for every call. Any program which calls this function must also call `gs_close_vb_timer` to remove the timer function from the system's vertical blank interrupt server chain.

See also `gs_close_vb_timer`, `gs_vb_time`, `gs_vb_timer_reset`

Function **gs_plot**
Plot a point in a bitmap

Synopsis

```
prev = gs_plot(bitmap,x, y,color);
d0          a0    d0 d1 d2

int prev;                /* previous color value */
struct BitMap *bitmap;   /* pointer to bitmap structure*/
int x,y;                 /* X,Y coordinate to plot */
int color;               /* color number to plot */
```

Libraries none

Description This function plots a point on the supplied bitmap in the specified color. It is a replacement for the sluggish `WritePixel` routine found in the system graphics library, and is many times faster. The only difference is that `gs_plot` accepts a pointer to a `BitMap` struct instead of a `RastPort` struct, and the color to plot is passed as an argument instead of using the primary drawing pen. The color number passed and returned is the color register number, or bitplane mask, of the pixel. In other words, plotting to a bitmap with a depth of 5 bitplanes allows 32 possible colors, numbered 0 through 31.

WARNING: for speed, this function does not check the bounds of the bitmap when plotting the point. It is the programmer's responsibility to make sure that the X,Y coordinate passed to this routine is within the bounds of the bitmap. For instance, if you have a bitmap that is 320 pixels wide, then valid X coordinates range from 0 to 319.

Returns A value is returned that is equal to the color number of the pixel at the X,Y coordinate before it is changed to the new color.

See also `gs_plot_reverse`, `gs_plot_test`

Function **gs_plot_reverse**
Complement a point in a bitmap

Synopsis

```
prey = gs_plot_reverse(bitmap, x, y);
d0          a0    d0 d1

int prev;                /* previous color value */
struct BitMap *bitmap;   /* pointer to bitmap structure*/
int x,y;                 /* X,Y coordinate to plot */
```

Libraries none

Description This function flips the color bits at the specified location in the bitmap, creating its complement. For instance if the color value at point X,Y of a 5 bitplane bitmap is 12 before calling this routine, it will be 19 upon return.

BEFORE 01100 binary, or 0C hex
AFTER 10011 binary, or 13 hex

WARNING: for speed, this function does not check the bounds of the bitmap when plotting the point.' It is the programmer's responsibility to make sure that the X,Y coordinate passed to this routine is within the bounds of the bitmap. For instance, if you have a bitmap that is 320 pixels wide, then valid X coordinates range from 0 to 319.

Returns A value is returned that is equal to the color number of the pixel at the X,Y coordinate before it is complemented.

See also `gs_plot, gs_plot_test`

Function **gs_plot_test**
Test the color of a pixel

Synopsis `color = gs_plot_test(bitmap, x, y);`
 `d0                    a0      d0 d1`

```
int color;                    /* pixel color */
struct BitMap *bitmap;       /* pointer to bitmap structure */
int x,y;                     /* X,Y coordinate of pixel */
```

Libraries none

Description This function gets the color value of a pixel in a bitmap. The color corresponds to a color register, or bitplane mask, for the pixel at the X,Y coordinates. For instance; in a bitmap with a depth of 5 bitplanes, color numbers may range from 0 to 31.

Returns A value is returned that is equal to the color number of the pixel at the X,Y coordinate.

See also `gs_plot, gs_plot_reverse`

Function **gs_random**
Generate a random number

Synopsis `n = gs_random(range);`
 `d0                    d0`

```
unsigned short n; /* 16 bit random number */
int range;        /* range, from 0 to 65535 if range == 0, then seed
generator */
```

Libraries none

Description This function returns a pseudorandom number in the range of 0 to (range-1). Note that if range is equal to 0, then a new seed value is taken from the system clock for use in subsequent calls to `gs_random`. In this case 0 is also returned.

This function will automatically seed itself from the system clock the first time you call it, so you should never have to reseed it.

Returns This function returns a 16 bit unsigned value as noted above.

Example

```

/*
Demonstrate example of getting a ,random X,Y
coordinate for possible object or pixel placement,
and casting a random die roll.
*/

#define SCREEN_WIDTH 640 /* typical hires screen width*/
#define SCREEN_HEIGHT 400 /*typical NTSC interlaced screen height */

unsigned short x,y,roll;
.
.
x = gs_random(SCREEN_WIDTH);/*get random x coordinate */
y = gs_random(SCREEN_HEIGHT);/*get random y coordinate */
.
.
/* simulate roll of 20 sided die */
roll = gs_random(20)+1; /* roll == 1 through 20*/

```

Function **gs_re_dim**
Specify a smaller real space than the entire bitmap area

Synopsis int status = gs_re_dim(int dlhand ,int width,int height);

```

int dlhand; /* display list handle */
int width;       /* pixel width of real space (<= bitmap.BytesPerRow*8) */
int height; /* pixel height of real space (<= bitmap.Rows) */

```

Description This function allows the user to specify a smaller real space than the entire bitmap area. When scrolling a bitmap larger than the display, there is usually no need to actually draw objects that aren't visible, even if they still reside in the normal real space bitmap area. See the aqua demo for an example of how this function is used.

When you call `gs_init_anim()`, the default real space dimensions are set to the entire bitmap area.

Returns 0 if successful
-1 if one of the new dimensions were larger than the bitmap.

Function **gs_remove_anim**
Remove an anim object and all attached children from the display lists

Synopsis void gs_remove_anim(anim);

```

                                         a0

struct anim struct *anim; /* pointer to anim structure */

```

Libraries none

Description This function removes an active anim object and all of its attached objects (if any) from the display lists. Removal is not immediate, however. In a single buffered environment, the next call to `gs_draw_anims` will remove it from the animation system. In a double buffered environment, two calls to `gs_draw_anims` will be required to totally remove the object. The call(s) to `gs_draw_anims` are necessary for cleanup operations such as restoring the background data beneath the object (if required), and removing it from active status.

It is not necessary to remove objects from the animation system before freeing save areas, freeing objects, and exiting your program.

See also `gs_add_anim, gs_draw_anims`

Function **gs_remove_cplx**
Remove an anim complex and all attached `*/` child anims from the display lists

Synopsis

```
void gs_remove_cplx(cplx);  
                a0  
  
struct anim cplx *cplx; /* pointer to anim complex */
```

Libraries none

Description This function removes an active anim complex and all of its attached anims (if any) from the display lists. Removal is not immediate, however. In a single buffered environment, the next call to `gs_draw_anims` will remove it from the animation system. In a double buffered environment, two calls to `gs_draw_anims` will be required to totally remove the object. The call(s) to `gs_draw_anims` are necessary for cleanup operations such as restoring the background data beneath the object (if required), and removing it from active status.

It is not necessary to remove objects from the animation system before freeing save areas, freeing objects, and exiting your program.

See also `gs_add_anim_cplx, gs_draw_anims`

Function **gs_remove_display**
Remove the custom view display

Synopsis

```
void gs_remove_display(display);  
  
struct display_struct *display; /* pointer to display to remove */
```

Libraries GRAPHICS

Description This function removes the custom GameSmith view display, and reloads the view previously displayed before the call to `gs_display` or `gs_create_display` was made. All display resources are freed. Note that this function will only free the display bitmap(s) if one of the GameSmith display functions allocated them.

See also `gs_display, gs_create_display`

Function **gs_remove_vb_server**
Remove a function from the vertical blank server chain

Synopsis

```
void gs_remove_vb_server(server);  
                a1  
  
struct Interrupt *server;
```

Libraries none

<i>Description</i>	This function removes a function from the vertical blank server chain that was previously placed there by a call to <code>gs_add_vb_server()</code> .
<i>See also</i>	<code>gs_add_vb_server</code>
<i>Function</i>	gs_replace_ucop Replace the current user copper list
<i>Synopsis</i>	<pre>void gs_replace_ucop(d, vp, hc); struct display_struct *d; /* pointer to display to affect */ struct gs_viewport *vp; /* pointer to viewport with which to swap user copper lists */ struct hard_copper *hc; /* pointer to hard copper structure which contains the new list(s) */</pre>
<i>Description</i>	This function replaces the current user copper list(s) for the specified viewport with the list(s) contained in the <code>hard_copper</code> structure. You may replace user copper lists at any time within you program. NOTE: This function merely SWAPS the lists. Nothing is freed or de-allocated. The purpose of a swap is two fold: a) it is very fast, and b) it takes into account the possibility that the user may want to reuse the list at a future time within his/her program. To free the list(s) returned (even if you didn't initially set up a user copper list for the viewport) in the <code>hard_copper</code> structure, you must call <code>gs_free_hard_copper()</code>.
<i>See also</i>	<code>gs_free_hard_copper()</code>
<i>Function</i>	gs_restore_anim_bg Restore any saved background data behind active anims
<i>Synopsis</i>	<pre>void gs_restore_anim_bg(handle); d0 int handle; /* display list handle */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	Sometimes you may need to do your own drawing to a bitmap, and you will want those results to appear behind anim objects in the bitmap. To accomplish this, you will need to use a <code>gs_restore_anim_bg/gs_draw_anim_objs</code> combination. Between these two function calls you will be able to do your own drawing (lines, block copies, or whatever). Note that the next function call to the animation system after this one MUST be <code>gs_draw_anim_objs</code>. This function tries to remove all active objects in the display list from the current drawing bitmap. Removal is the opposite of the method used to copy the image to the bitmap. That is, if background data was saved before the copy (<code>ANIM_SAVE_BG</code> flag set), then it is restored by this function, thus removing the image from display. If, however, the <code>ANIM_CLEAR</code> flag is set instead of the <code>ANIM_SAVE_BG</code> flag, then the image is simply removed from the bitmap in the same manner it was copied to it (mask or no mask); clearing the bitplane areas affected by the image to zero. If neither flag is set in the anim structure, then this function has no effect on those anims, and the anim remains in the bitmap (the anim. will leave a trail if moved).

A `gs_restore_anim_bg/gs_draw_anim_objs` combination has exactly the same effect as a single call to `gs_draw_anims`.

See also `gs_draw_anim_objs`

Function **gs_reverse_anim**
Reverse the animation direction of an anim object

Synopsis

```
void gs_reverse_anim(anim);
                        a0

struct anim_struct *anim; /* pointer to anim structure */
```

Libraries none

Description This function reverses the animation direction of an anim object. That is, if the object had been animating forward (the default), then it will start animating backward, or vice versa. The next call to a function which animates the object will reflect the change.

See also `gs_anim_forward`, `gs_anim_backward`, `gs_anim_obj`, `gs_anim_parent`,
`gs_anim_children`

Function **gs_reverse_cplx**
Reverse the animation direction of an anim complex

Synopsis

```
void gs_reverse_cplx(cplx);
                        a0

struct anim_cplx *cplx; /* pointer to anim complex */
```

Libraries none

Description This function reverses the animation direction of the current animation sequence of an anim complex. That is, if the object had been animating forward (the default), then it will start animating backward, or vice versa. The next call to a function which animates the object will reflect the change.

Note that changing the animation sequence will reset the animation direction to the default of forward.

See also `gs_cplx_forward`, `gs_cplx_backward`, `gs_anim_cplx`, `gs_anim_cplx_parent`,
`gs_anim_cplx_children`

Function **gs_rs_offset**
Define real space offset of a bitmap in virtual space

Synopsis

```
void gs_rs_offset(handle, x, y);
                  d0    d1 d2

int handle; /* display list handle */
int x,y;    /* X,Y coordinates of bitmap */
```

Libraries none

Description This function allows you to specify your bitmap offset within the allowable 32K virtual space. This means that you may scroll your bitmap around in virtual space, and anim objects will automatically be drawn in the bitmap(s) if they fall within the real space bounds (with a call to `gs_draw_anims` of course). Valid X offsets range from 0 (default) to 32768 - (bitmap pixel width). Valid Y offsets range from 0 (default) to 32768 - (bitmap pixel height).

See also `gs_get_display_list`

Function **gs_rs_window**
Shift a real space window which is smaller than the bitmap area to any position over the bitmap

Synopsis `void gs_rs_window(int hand, int x_off, int y_off);`

```
int hand;    /* display list handle */
int x_off;   /* offset in pixels of left edge of real space window over
the bitmap area. */
int y_off;   /* offset in pixels of top edge of real space window over
the bitmap. */
```

Description This function allows the user to shift the real space window which is smaller than the bitmap area to any position over the bitmap. This is handy when scrolling a bitmap and yet keeping the actual real space window surrounding the actual display. See the aqua demo for a good example of how to use this function.

WARNING!!!!!!! FOR SPEED PURPOSES, THIS FUNCTION DOES >NOT< CHECK THE X AND Y INPUT PARAMETERS FOR VALIDITY. IF YOU ALLOW YOUR REAL SPACE WINDOW TO FALL OUTSIDE OF THE ACTUAL BITMAP AREA, OBJECTS MAY BE COPIED OUTSIDE OF THE BITMAP AND THUS CORRUPT MEMORY!

When you call `gs_init_anim()`, the left & top default offsets are set to zero (real space is also, by default, the entire bitmap area).

Function **gs_scan_copper**
Rescan a GameSmith display and incorporate new copper changes

Synopsis `status = gs_scan_copper(display);`

```
int status;
struct display_struct *display;
```

Libraries GRAPHICS

Description This function is necessary to incorporate changes to the copper list of a GameSmith display caused by a `gs_user_copper` function call. Note that any color palette changes previously made with `gs_LoadRGB` or `gs_SetRGB` will be lost.

Returns If the rescan is successful, zero is returned. Nonzero indicates not enough memory was available for new copper lists.

See also `gs_user_copper`

Function **gs_scroll_vp**
Scroll a GameSmith viewport

Synopsis

```
status =gs_scroll_vp(display, viewport , deltax, deltay, sync);

int status;          /* return status */
struct display_struct *display; /* pointer to display struct */
int viewport;        /* viewport number (0 to n) */
int deltax;          /* pixel amount to shift on the X axis */
int deltay;          /* pixel amount to shift on the Y axis */
int sync;            /* flag for synchronized display update*/
```

Libraries **GRAPHICS**

Description

This function allows your program to scroll a GameSmith viewport which has a bitmap larger than the viewport display window. The viewport number is a value from 0 (topmost and first in the viewport chain) to n (bottom most and last in the viewport chain). The values for delta X and delta Y can be positive or negative. Positive X values scroll the viewport to the right over the bitmap, and positive Y values scroll the viewport down over the bitmap. A nonzero sync value allows changes to synchronize with the next vertical blank period, avoiding visible display disturbances. This method is recommended.

If you think of looking at a newspaper through a magnifying glass, you might get a better idea of how viewports and bitmaps work. The magnifying glass is the viewport, and the newspaper is the bitmap. Moving the magnifying glass shifts your view of the newspaper, just as scrolling the viewport shifts your view of the bitmap.

Note that this function only has effect on bitmaps which are larger than their respective display viewports, and the display in which the viewport resides should have the GSV_SCROLLABLE bit set.

ASSEMBLER PROGRAMMERS: This function was written in 'C', and requires that you place all input parameters on the stack before making the call.

Returns

A successful completion is indicated by a zero status. Otherwise, possible return values are as follows. Note that for informational bits, your program must perform a bitwise AND operation (&) to determine if the appropriate bits are set in the return status.

Scroll a GameSmith viewport

Errors:

GSVP_NOVP	Invalid viewport number
-----------	-------------------------

Information bits (if no error):

GSVP_PF1_LEFT	Playfield 1 has reached the left edge of bitmap
GSVP_PF1_RIGHT	Playfield 1 has reached the right edge of bitmap
GSVP_PF1_TOP	Playfield 1 has reached the top edge of bitmap
GSVP_PF1_BOTTOM	Playfield 1 has reached the bottom edge of bitmap
GSVP_PF2_LEFT	Playfield 2 has reached the left edge of bitmap
GSVP_PF2_RIGHT	Playfield 2 has reached the right edge of bitmap
GSVP_PF2_TOP	Playfield 2 has reached the top edge of bitmap
GSVP_PF2_BOTTOM	Playfield 2 has reached the bottom edge of bitmap

See also

gs_create_display, gs_scroll_vp_pf1, gs_scroll_vp_pf2

Function

gs_scroll_vp_pf1

Scroll playfield 1 of a GameSmith viewport

Synopsis

```
status=gs_scroll_vp_pf1(display,viewport,deltax,delay, sync);

int status;          /* return status */
struct display struct *display; /* pointer to display struct */
int viewport;        /* viewport number (0 to n) */
int deltax;          /* pixel amount to shift on the X axis */
int delay;           /* pixel amount to shift on the Y axis */
int sync;            /* flag for synchronized display update */
```

Libraries GRAPHICS

Description This function allows your program to scroll playfield 1 of a GameSmith viewport which has a bitmap larger than the viewport display window. Playfield 1 consists of bitplanes 0, 2, 4, etc., that make up the viewport display. Typically, you will use this function on a viewport that is using the Amiga's dual playfield mode. The viewport number is a value from 0 (topmost and first in the viewport chain) to n (bottom most and last in the viewport chain). The values for delta X and delta Y can be positive or negative. Positive X values scroll the viewport to the right over the bitmap, and positive Y values scroll the viewport down over the bitmap. A nonzero sync value allows changes to synchronize with the next vertical blank period, avoiding visible display disturbances. This method is recommended. Note that this function only has effect on bitmaps which are larger than their respective display viewports, and the display in which the viewport resides should have the GSV_SCROLLABLE bit set.

ASSEMBLER PROGRAMMERS: This function was written in 'C', and requires that you place all input parameters on the stack before making the call.

Returns A successful completion is indicated by a zero status. Otherwise, possible return values are as follows. Note that for informational bits, your program must perform a bitwise AND operation (&) to determine if the appropriate bits are set in the return status.

Errors:

GSVP_NOVP Invalid viewport number

Information bits (if no error)

GSVP_PF1_LEFT	Playfield 1 has reached the left edge of bitmap
GSVP_PF1_RIGHT	Playfield 1 has reached the right edge of bitmap
GSVP_PF1_TOP	Playfield 1 has reached the top edge of bitmap
GSVP_PF1_BOTTOM	Playfield 1 has reached the bottom edge of bitmap

See also gs_create_display, gs_scroll_vp, gs_scroll_vp_pf2

Function **gs_scroll_vp_pf2**
Scroll playfield 2 of a GameSmith viewport

Synopsis

```
status=gs_scroll_vp_pf2(display,viewport,deltax,delay, sync);

int status;          /* return status */
struct display struct *display; /* pointer to display struct */
int viewport;        /* viewport number (0 to n) */
int deltax;          /* pixel amount to shift on the X axis */
int delay;           /* pixel amount to shift on the Y axis */
int sync;            /* flag for synchronized display update */
```

Libraries GRAPHICS

Description This function allows your program to scroll playfield 2 of a GameSmith viewport which has a bitmap larger than the viewport display window. Playfield 2 consists of bitplanes 1, 3, 5, etc., that make up the viewport display. Typically, you will use this function on a viewport that is using the Amiga's dual playfield mode. The viewport number is a value from 0 (topmost and first in the viewport chain) to n (bottom most and last in the viewport chain). The values for delta X and delta Y can be positive or negative. Positive X values scroll the viewport to the right over the bitmap, and positive Y values scroll the viewport down over the bitmap. A nonzero sync value allows changes to synchronize with the next vertical blank period, avoiding visible display disturbances. This method is recommended. Note that this function only has effect on bitmaps which are larger than their respective display viewports, and the display in which the viewport resides should have the GSV_SCROLLABLE bit set.

ASSEMBLER PROGRAMMERS: This function was written in 'C', and requires that you place all input parameters on the stack before making the call.

Returns A successful completion is indicated by a zero status. Otherwise, possible return values are as follows. Note that for informational bits, your program must perform a bitwise AND operation (&) to determine if the appropriate bits are set in the return status.

Errors:

GSVP_NOVP Invalid viewport number

Information bits (if no error)

GSVP_PF2_LEFT	Playfield 2 has reached the left edge of bitmap
GSVP_PF2_RIGHT	Playfield 2 has reached the right edge of bitmap
GSVP_PF2_TOP	Playfield 2 has reached the top edge of bitmap
GSVP_PF2_BOTTOM	Playfield 2 has reached the bottom edge of bitmap

See also `gs_create_display`, `gs_scroll_vp`, `gs_scroll_vp_pf1`

Function **`gs_set_anim_bounds`**
Define the bounds for all objects in a display list.

Synopsis

```
void gs_set_anim_bounds(handle, x1, y1, x2, y2);
                        d0      d1  d2  d3  d4

int handle; /* display list handle */
int x1;     /* leftmost usable offset within virtual space */
int y1;     /* top most usable offset */
int x2;     /* right most usable offset */
int y2;     /* bottom most usable offset */
```

Libraries none

Description By default, objects in a display list are free to roam around the entire 32K virtual space, and bounds checking is enabled. You may further confine object movement with this function. Note that valid bounds values range from 0 to 32727 (0x7fff hex). This function may be called at any time to change the anim bounds as desired.

After setting the animation bounds, anim objects will have the appropriate flag bit set should they attempt to cross one or more borders. In this event the current image is placed at the border's edge. Refer to the User's Guide for more information on bounds checking.

Objects which fall partially or wholly outside real space (the bitmap area) are simply not drawn. Remember that the ANIM_VCOLLIDE bit is available if you want to allow object-to-object collisions outside of real space.

See also `gs_disable_anim_bounds`, `gs_rs_offset`

Function **gs_set_anim_cell**
Select image to be displayed

Synopsis

```
void gs_set_anim_cell(anim, cell);
                        a0    d0

struct anim_struct *anim; /* pointer to anim structure */
int cell;                 /* cell number to display (0 - n) */
```

Libraries none

Description This function allows you to choose which image will be displayed from an animation sequence with the next call to `gs_draw_anims`. The cell, or frame number, begins at 0 for the first cell in an animation sequence. If your program specifies a cell that does not exist in the anim, the last cell in the anim is used for the next display. To find out how many cells exist in a particular animation sequence, check the count field of the anim structure. The anim object must be active in the animation system for this function to have any effect.

See also `gs_add_anim`

Function **gs_set_anim_flicker**
Cause an active anim object to flicker

Synopsis

```
void gs_set_anim_flicker (anim);
                        a0

struct anim_struct *anim; /* pointer to anim structure */
```

Libraries none

Description This function makes an anim object and all of its attached anims (if any) appear to flicker when displayed. This is accomplished by only displaying the object(s) in the first bitmap of a double buffered display. Needless to say, this has no effect on objects in a single buffered display environment. The speed at which the object(s) appear to flicker is entirely up to the speed at which your program flips between the two display bitmaps.

A common visual effect in games is to cause the controllable character to flicker when he or she is just coming to life on the screen. This gives the player a chance to get his or her bearings before starting game play. It is also common in this situation to turn off all collision detection for the controllable character.

The anim object must be active in the animation system for this function to have any effect.

See also `gs_clear_anim_flicker`, `gs_flip_display`, `gs_next_anim_page`,
`gs_disable_anim_collision`

Function **gs_set_anim_merge**
Turn on merge mode for an anim object

Synopsis

```
void gs_set_anim_merge(anim);
                        a0

struct anim_struct *anim; /* pointer to anim structure */
```

Description This function turns on merge mode for the specified anim object and all of its attached anims (if any). This allows various types of translucency effects. Note that anims created in CITAS with the TRANSLUCENCY option will automatically have the merge mode turned on.

When the merge flag is set in an anim object, unaffected bitplanes will not be cleared when the object is displayed. That is, if an image has a planes field of 0x06 (hex 6), and a depth of 3, bitplane 0 will NOT be cleared in the image area when the image is copied to a bitmap. Normally, unaffected bitplanes will be cleared so that the correct colors are shown for the image.

The anim object need not be active in the animation system before calling this function.

See also `gs_clear_anim_merge`

Function **gs_set_collision**
Set up an object-to-object collision handler for use with a display list.

Synopsis

```
void gs_set_collision(handle, function);
                        d0      a0

int handle; /* display list handle */
void *function; /* pointer to collision handler function */
```

Libraries none

Description Through this function, you supply the animation system with a function to be called when objects in a display list collide with each other. Your function can handle object collisions in any way you want it to. If you do not supply a collision handler, then object collisions are simply ignored. As stated in the User's Guide, 'C' functions MUST be standard ANSI functions that accept their arguments on the stack. Because of a small amount of extra setup work, calling 'C' collision handlers is slightly slower than calling collision handlers written in assembler.

Example

```
/*
Set up a collision handler
*/

void setup()
{
gs_set_collision(dlist,&collision);
```


Description This function allows you to choose which image will be displayed from the current animation sequence in an anim complex with the next call to `gs_draw_anims`. The cell, or frame number, begins at 0 for the first cell in an animation sequence. If your program specifies a cell that does not exist in the anim, the last cell in the anim is used for the next display. To find out how many cells exist in a particular animation sequence, check the count field of the anim structure.

The anim complex must be active in the animation system for this function to have any effect.

See also `gs_add_anim_cplx`

Function **gs_set_cplx_flicker**
Cause an active anim complex to flicker

Synopsis

```
void gs_set_cplx_flicker(cplx);
                        a0

struct anim cplx *cplx; /* pointer to anim complex */
```

Libraries none

Description This function makes an anim complex and all of its attached anims (if any) appear to flicker when displayed. This is accomplished by only displaying the object(s) in the first bitmap of a double buffered display. Needless to say, this has no effect on objects in a single buffered display environment. The speed at which the object(s) appear to flicker is entirely up to the speed at which your program flips between the two display bitmaps.

A common visual effect in games is to cause the controllable character to flicker when he or she is just coming to life on the screen. This gives the player a chance to get his or her bearings before starting game play. It is also common in this situation to turn off all collision detection for the controllable character.

The anim complex must be active in the animation system for this function to have any effect.

See also `gs_clear_cplx_flicker`, `gs_flip_display`, `gs_next_anim_page`,
`gs_disable_cplx_collision`

Function **gs_set_cplx_merge**
Turn on merge mode for an anim complex

Synopsis

```
void gs_set_cplx_merge(cplx);
                        a0

struct anim cplx *cplx; /* pointer to anim complex */
```

Libraries none

Description This function turns on merge mode for all anims in an anim complex and all attached anims (if any). This allows various types of translucency effects. Note that anims created in CITAS with the TRANSLUCENCY option will automatically have the merge mode turned on.

When the merge flag is set in an anim object, unaffected bitplanes will not be cleared when the object is displayed. That is, if an image has a planes field of 0x06 (hex 6), and a depth of 3, bitplane 0 will NOT be cleared in the image area when the image is copied to a bitmap. Normally, unaffected bitplanes will be cleared so that the correct colors are shown for the image.

The anim complex need not be active in the animation system before calling this function.

See also `gs_clear_cplx_merge`

Function **gs_set_cplx_prio**
Set the display priority for an anim complex

Synopsis

```
void gs_set_cplx_prio(cplx, prio);
                        a0      d0

struct anim cplx *cplx; /* pointer to anim complex */
int prio;               /* new display priority for complex */
```

Libraries none

Description This function sets the display priority for all anims in an anim complex. Any attached anims are unaffected. Note that the new priority will not take effect in active objects until you call the appropriate sort function.

See also `gs_sort_anims`

Function **gs_set_cplx_seq**
Select an animation sequence for display

Synopsis

```
status = gs_set_cplx_seq(cplx, seq, x, y);
d0                      a0      d0 d1 d2

int status;              /* return status */
struct anim cplx *cplx; /* pointer to anim complex */
int seq;                 /* anim sequence number (0 - n) */
int x,y;                 /* X,Y coordinates for object placement*/
```

Libraries none

Description This function allows your program to select which animation sequence (anim object) will be used for display. Only one animation sequence in an anim complex can be displayed at a time. The sequence number starts with 0 for the first anim in the complex. Typically, you would select the sequence number through the use of a 'C' defined or an assembler equated name such as MAN_LEFT, or MAN_RIGHT (CITAS makes such names available to your program).

The new sequence will be placed at the supplied X,Y coordinates with the next call to `gs_draw_anims`. Note that the cell number for the specified anim sequence is NOT reset by this call.

The animation direction for the complex is reset to forward as a result of this call, regardless of whether or not it had been animating backward.

Returns A successful completion is indicated by a return of 0. Nonzero is returned in the event of an error, meaning that you specified a sequence number that does not exist in the anim complex. In this event, the sequence number is not changed. To determine the number of anim sequences, check the cnt field of the anim complex structure.

See also `gs_set_cplx_cell`, `gs_cplx_backward`

Function **gs_set_period**
Set the period for a sound channel

Synopsis

```
void gs_set_period(period, channel);
                   d0      d1

int period; /* new period */
int channel; /* channel to change */
```

Libraries none

Description This function sets a new period for the specified channel. The channel must be one of CHANNEL0, CHANNEL1, CHANNEL2, or CHANNEL3. This function only affects the period of a sound channel that is currently playing a sound sample. The initial period is set per the controlling sound structure.

See also `gs_set_volume`

Function **gs_set_volume**
Set the volume for a sound channel

Synopsis

```
void gs_set_volume(volume, channel );
                   d0      d1

int volume; /* new volume */
int channel; /* channel to change */
```

Libraries none

Description This function sets a new volume for the specified channel. The channel must be one of CHANNEL0, CHANNEL1, CHANNEL2, or CHANNEL3. Valid volume values range from 0 to 64, with 64 being the loudest. This function only affects the volume of a sound channel that is currently playing a sound sample. The initial volume is set per the controlling sound structure.

See also `gs_set_period`

Function **gs_SetRGB**
Reload a single color palette entry of GameSmith viewport

Synopsis

```
status = gs_SetRGB(display, vp, color_num, color);

int status;
struct display_struct *display;
int vp; /* viewport number to modify */
int color_num; /* color number to change (0-n) */
unsigned long color; /* new color value */
```

Libraries GRAPHICS

<i>Description</i>	This function reloads a single color palette entry of a GameSmith viewport. The viewport number is a value from 0 (first and topmost in the display) to n (last and bottommost in the display). The color number is a value from 0 to n, corresponding to the correct entry to be modified in the palette. For instance, a 5 bitplane display contains 32 colors. Color numbers range from 0 to 31. The color value must be in standard GameSmith format.
--------------------	---

ASSEMBLER PROGRAMMERS: This function was written in ‘C’, and therefore requires that all input parameters be passed on the stack. See the User’s Guide for more information about calling ‘C’ functions from assembler.

Returns If the viewport number is found, then the color palette entry is modified. Otherwise the value GSVP NOVP is returned.

See also `gs_create_display`, `gs_LoadRGB`

<i>Function</i>	gs_show_display Show a ViewPort display
-----------------	---

```
Synopsis    void gs_show_display(display, sync);

           struct display struct *display; /* pointer to display to show */
           int sync;    /* flag for syncing with vertical blank */
```

Libraries GRAPHICS

Description This function calls the systems graphics function LoadView with the appropriate custom view previously set up by gs_display or gs_create_display. For single buffered displays, there is only one view that can be loaded. For double buffered displays, the currently active view is loaded (see `gs_flip_display`).

This is useful, for instance, if the programmer allows the ability to switch back to an Intuition screen, or possibly some other view, and then wants to return to the custom GameSmith view.

The input parameter `sync`, if nonzero, tells the function to wait until the next vertical blank period before displaying the view. This prevents any display flicker. If `sync` is zero, then the view is immediately loaded, which may cause display flicker.

ASSEMBLER PROGRAMMERS: This function is written in 'C', and requires that the sync value (32 bit long word) be placed on the stack before making the call.

See also `gs_display`, `gs_create_display`, `gs_flip_display`, `gs_old_display`,
`gs_remove_display`

<i>Function</i>	gs_sort_anims
	Sort display priorities for objects in a display list

```
Synopsis      void    gs_sort_anims(handle);
                                d0

                int handle; /* display list handle */
```

<i>Libraries</i>	none
<i>Description</i>	This function sorts all anim objects in a display list according to display priority. Display priorities range from 0 to 65535 (0xffff hex). Objects with a lower priority are copied to the bitmap before higher numbered objects, allowing objects with a higher priority to appear on top of objects with a lower priority. To change the display priority of a simple anim object, merely change the value in the prio field of the anim structure and call this function. To change the priority of an anim complex, call the function <code>gs_set_cplx_prio</code> and then call this function. Display priorities only take effect when adding an object to a display list or when resorting them.
<i>See also</i>	<code>gs_set_cplx_prio</code> , <code>gs_get_display_list</code>
<i>Function</i>	gs_sound_check Get the status of the sound channels
<i>Synopsis</i>	<pre>status = gs_sound_check(void); d0 int status; /* status information */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function reads the current status of the sound channels. The status return value contains valid information in the lower 16 bits. Bits 0 - 7 contain the busy status of the sound channels, and bits 8 - 15 contain start request information (sounds that are scheduled to begin at the next vertical blank interrupt). The bits in each byte correspond to the bit definitions for CHANNEL0, CHANNEL1, CHANNEL2, and CHANNEL3. When no sound samples are playing, and no sounds are scheduled to begin, the contents of status will be zero.
<i>Example</i>	To determine if CHANNEL1 is busy, simply perform a logical AND operation on the status return. If the value is nonzero (the channel busy bit is set), then the channel is busy playing a sound. <pre>status = gs_sound_check(); if (status & CHANNEL1) printf("channel 1 is busy");</pre> To determine if CHANNEL1 is scheduled to have a sound begin, you first need to shift the contents of status to the right by 8 bits before performing the AND operation. <pre>if ((status>>8) & CHANNEL1) printf("channel 1 is scheduled to play a sound");</pre>
<i>See also</i>	<code>gs_start_sound</code> , <code>gs_stop_sound</code>
<i>Function</i>	gs_start_sound Start a sound sample playing

<i>Synopsis</i>	<pre>void gs_start_sound(sound,channel); a1 d0 struct sound_struct *sound; int channel; /* channel to play sound on */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function schedules a sound sample to begin playing on the specified channel when the next vertical blank interval occurs. If the channel is busy, then an additional vertical blank interval is required to stop the currently playing sound and start the new one. The channel must be one of CHANNEL0, CHANNEL1, CHANNEL2, or CHANNEL3. Only one channel may be started at a time. To play a sample on more than one channel at the same time, place an Enable()/Disable() function call pair around your code which starts the sample playing. Note that at this time, the only valid sound type is SND_EFX. Make sure that the type field of the sound structure is set accordingly.
<i>See also</i>	gs_stop_sound, gs_set_volume, gs_set_period, gs_open_sound, gs_sound_check
<i>Function</i>	gs_step_vector Generate the next set of X,Y coordinates along a line
<i>Synopsis</i>	<pre>status = gs_step_vector(vector,step_rate,x,y); d0 d1 int status; int vector; /* vector number (0 - 99) */ int step_rate; /* increment precision (1 - n) */ int *x; /* pointer to new X coordinate */ int *y; /* pointer to new Y coordinate */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function increments the current X and Y coordinate of the specified vector by step_rate amount, and returns the new values in the integers pointed to by x and y. The step rate may be any value from 1 to n, and specifies the precision by which you would like the new coordinates incremented. Generally, the precision value applies to pixels on the screen, so in order to increment one pixel along the line, the step rate should be 1. This is the smallest precision available. ASSEMBLER PROGRAMMERS: note that there is no register specification for return values of x and y. This is because from assembler, the function operates slightly differently by returning the new X and Y coordinates in registers d0 and d1 respectively.
<i>Returns</i>	As specified above, the new X and Y coordinates are returned in the integers pointed to by x and y (for 'C' programmers), or in registers d0 and d1 (for assembler programmers). When the new X and Y coordinate is equal to or greater than the ending coordinate of the line, a value of 0,0 is returned (both X and Y contain zero). This will hold true until the vector is reinitialized. When the end of the line is reached, status will be zero.
<i>See also</i>	gs_init_vector, gs_ustep_vector

<i>Function</i>	gs_stop_sound Stop all sound on a given sound channel
<i>Synopsis</i>	<pre>void gs_stop_sound(channel); d0 int channel; /* channel to stop */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function stops all sound on the specified channel. The channel must be one of CHANNEL0, CHANNEL1, CHANNEL2, or CHANNEL3. Only one channel may be stopped at a time. Unlike the function <code>gs_start_sound</code> which schedules a sound event, this function stops the specified channel immediately. Both the busy flags and the start request flags are cleared (see <code>gs_sound_check</code>). This means that any sound that is currently playing is aborted, as well as any sound that was just scheduled to play.
<i>See also</i>	<code>gs_start_sound</code> , <code>gs_set_volume</code> , <code>gs_set_period</code> , <code>gs_open_sound</code> , <code>gs_sound_check</code>
<i>Function</i>	gs_task_prio Set the current task priority
<i>Synopsis</i>	<pre>old_priority = gs_task_prio(new_priority); d0 d0 int old_priority; /* old priority value */ int new_priority; /* new priority to assign */</pre>
<i>Libraries</i>	none
<i>Description</i>	This function assigns a new priority to your program's task.
<i>Returns</i>	The old task priority is returned. Your program should ALWAYS set the task priority back to the old priority before it exits. Not doing so could cause serious system problems if the program was started from the command line interface, and can even lead to system crash.
<i>Function</i>	gs_user_copper Set up or change a custom copper list
<i>Synopsis</i>	<pre>status = gs_user_copper(copper); int status; struct copper_struct *copper; /* pointer to copper struct */</pre>
<i>Libraries</i>	GRAPHICS
<i>Description</i>	This function allows an easy means of setting up a custom copper list on any Amiga display. It can also be used to change an existing user copper list for dynamic effects. All of this is accomplished through a table of copper commands pointed to by the copper structure.

Copper Commands:

UC_MOVE	Move a value into a hardware register. The syntax for this is UC_MOVE,reg,value. The reg is the hardware offset for the register (such as 0x96 is the hardware offset for the DMA control register. Refer to the AMIGA Hardware Reference Manual for complete details). Value is the 16 bit value you want placed in the hardware register
UC_WAIT	Wait for the video beam to reach a specified Y,X coordinate before executing the next command in the list Syntax is UC_WAIT,y,x. Note that all X and Y values assume a low resolution display. Again, see the hardware manual for details.
UC_NOSPRITES	Turn off sprites. This is especially handy when using the GameSmith display system and you don't want that pesky mouse pointer on your screen. This command requires no additional values.
UC_SPRITES	Turn on sprites. This allows sprites to reappear. If you turn off sprites for a GameSmith display, you should re-enable them at the bottom of the display. This way the mouse pointer will reappear if you return control to the Workbench display.
UC_SETCOLOR	Set a color register. Syntax is UC_SETCOLOR, color number, color_value. Color values are in old 4 bit format (4 bits each for red,green, and blue). This can only affect color registers 0 through 31.
UC_END	This marks the end of the copper list. This command MUST be the last command in the table.

The Copper Structure:

```
struct copper_struct
{
    unsigned *short list; /* ptr to copper list */
    void *copctrl; /* ptr to UCopList */
    void *display; /* ptr to appropriate display struct */
};
```

list

This is a pointer to the list of copper commands.copctrl This is a pointer to a system UCopList structure. Normally, you will fill this field with NULL and never worry about it. Only when changing the copper list of an existing display do you need to fill this field. display If you are passing a copper list to the `gs_create_display` function, simply fill this field with NULL and ignore it Otherwise, if you are calling `gs_user_copper` yourself you will need to fill this field with the appropriate controlling structure for the display you're modifying. This will be either a pointer to an Intuition Screen struct, or a pointer to a ViewPort struct, depending on the type of display.

Modifying an Existing Display:

To modify an existing Intuition screen, follow these steps:

- Issue a `Forbid()` call.

- Set the list pointer of the copper structure to point to your command table.
- Set the copctrl pointer to that of the screen's UCopList structure. Copy the screen >ViewPort.UCopIns field.
- Set the display pointer of the copper structure to point to the Screen structure.
- Call this function with the UCFLAGS_INTUITION flag.
- Issue a Permit() call.
- Call RethinkDisplay() to show the change.

To modify an existing ViewPort

- Issue a Forbid() call.
- Set the list pointer of the copper structure to point to your command table.
- Set the copctrl pointer to that of the ViewPort's UCopList structure. Copy the viewport >UCopIns field.
- Set the display pointer of the copper structure to point to the ViewPort structure.
- Call this function with the UCFLAGS_VIEWPORT flag.
- Issue a Permit() call.
- Call MrgCop() to incorporate your new instruction into the display View
- Call LoadView() to show the change.
- If the display is a GameSmith display, call `gs_scan_copper()` and then `gs_show_display()`.

Because this function was written in 'C', assembler programmers will need to push the function parameters on the stack before making this call. Also, see the include file GameSmith.i for exact structure field names.

Returns

The status return will be zero if no error occurs. Otherwise, the following error codes are possible:

- 1 Insufficient memory to allocate a UCopList control structure
- 2 Invalid command in copper list

Example

```
/*
Modify the copper lists of a double buffered GameSmith display so that
the background color for the top half of the display is bright blue, and
the background color for the bottom half of the display is black. Note
that this could have been set up easier when calling gs_create_display.
This example assumes the programmer needs to change the colors at some
point after the initial display setup. */

struct display_struct *display; /* pointer to display */
struct gs_viewport *vp; /* viewport to modify */
unsigned short list[] =
{
    UC_NOSPRITES,          /* no sprites */
    UC_SETCOLOR,0,0x000f,  /* background color bright blue */
    UC_WAIT,127,0, /* wait for mid screen */
    UC_SETCOLOR,0,0, /* background color black */
    UC_WAIT,254,250, /* wait for bottom of screen */
    UC_END             /* end of copper list */
};

struct copper_struct copper
{
    list, /* address of copper list */
    NULL,
```

```

NULL
);

/*
The following assumes a display has already been set up by
gs_create_display or gs_display.
*/
Forbid(); /* make sure nothing else messes with list */
copper.copctrl=vp->viewport1->UCopIns;
copper.display=vp->viewport1;
gs_user_copper(&copper,UCFLAGS_VIEWPORT);
copper.copctrl=vp->viewport2->UCopIns;
copper.display=vp->viewport2;
gs_user_copper(&copper,UCFLAGS_VIEWPORT);
Permit();
MrgCop(display->view1);
MrgCop(display->view2);
LoadView(display->view1);
LoadView(display->view2);
gs_scan_copper(display); /* use new copper list */
gs_show_display(display,1); /* show new copper list */
gs_next_anim_page(dlist);

```

See also `gs_create_display`, `gs_scan_copper`

Function **gs_ustep_vector**
Generate the next set of X,Y coordinates along a line

Synopsis

```

status = gs_ustep_vector(vector,step_rate,x,y);
                        d0      d1

int status; int vector; /* vector number (0 - 99) */
int step_rate; /* increment precision (1 - n) */
int *x; /* pointer to new X coordinate */
int *y; /* pointer to new Y coordinate */

```

Libraries none

Description This function increments the current X and Y coordinate of the specified vector by `step_rate` amount, and returns the new values in the integers pointed to by `x` and `y`. The `step_rate` may be any value from 1 to `n`, and specifies the precision by which you would like the new coordinates incremented. Generally, the precision value applies to pixels on the screen, so in order to increment one pixel along the line, the `step_rate` should be 1. This is the smallest precision available.

This function differs from the function `gs_step_vector` in that movement is more uniform. If the vector routines will be used to draw lines, then generally `gs_step_vector` will be preferable. For object movement, this function should be used, as it provides less jagged movement.

ASSEMBLER PROGRAMMERS: note that there is no register specification for return values of `x` and `y`. This is because from assembler the function operates slightly differently by returning the new X and Y coordinates in registers `d0` and `d1` respectively.

Returns As specified above, the new X and Y coordinates are returned in the integers pointed to by `x` and `y` (for 'C' programmers), or in registers `d0` and `d1` (for assembler programmers). When the new X and Y coordinate is equal to or greater than the ending coordinate of the

line, a value of 0,0 is returned (both X and Y contain zero).

This will hold true until the vector is reinitialized. When the end of the line is reached, status will be zero.

See also `gs_init_vector`, `gs_step_vector`

Function **gs_vb_time**
Get the current value of the vertical blank timer

Synopsis

```
t = gs_vb_time(void);
d0

unsigned long t; /* timer value automatically incremented every vertical
blank interval */
```

Libraries none

Description This function polls the value of the custom vertical blank timer. After the timer routine is installed with `gs_open_vb_timer`, you may make this call at any point in your program to determine the current counter value.

This can be very useful, for instance, as a time delay within a game to guarantee that a program runs at (about) the same speed on any model of computer.

Returns An unsigned long integer is returned containing a value equal to the number of vertical blank intervals since last reset.

Example

```
/*
Event timer based on approximate amount of time required to perform a
certain task within a game.
```

```
NOTE: This example assumes a game which takes over the entire machine, as
it utilizes a busywait loop while waiting for time to elapse. A task with
a busy-wait loop eats up a lot of CPU time, and does not multitask well.
In order for a task to multitask well, an elegant solution might be to
allocate a signal, add a delay/ counter function to the vertical blank
server chain, and wait until the main program receives a signal from the
interrupt routine.
*/
```

```
#define DELAY_TIME 30 /* You have determined, from a previous test which
recorded 1000 typical loop times, that the least amount of time required
is 30 vertical blank intervals. */
```

```
main()
{
if (gs_open_libs(GRAPHICS,0)) /* open AmigaDOS libs */
exit(01);
gs_open_vb_timer();/* open & install timer */
gs_vb_timer_reset();/* make sure counter starts at 0 */
game_loop();/* call the main loop */
gs_close_vb_timer();/* close & remove timer */
gs_close_libs();/* close all opened Amiga libs */
}
```

```
void game_loop()
```

```

{
int not_done=1;

while (not_done)
{
/* call function to update graphics */
/* call function to handle sound */
/* call function to check user input */
/* BUSY WAIT LOOP! DOES NOT MULTITASK WELL! */
while (gs_vb_time() < DELAY_TIME); /* time      delay*/
gs_vb_timer_reset(); /*start counter over at 0*/
}
}

```

See also `gs_close_vb_timer,` `gs_open_vb_timer,` `gs_vb_timer_reset`

Function **gs_vb_timer_reset**
Reset the vertical blank timer count to 0

Synopsis `void gs_vb_timer_reset(void);`

Description This function resets the vertical blank timer count to zero. With the advent of the next vertical blank period, the count will increment to 1, and so on.

See also `gs_close_vb_timer,` `gs_open_vb_timer,` `gs_vb_time`

Function **gs_version**
Get the GameSmith library version number

Synopsis `version = gs_version(void);`
 `d0`

Description This function returns the version, release, and level number of the GameSmith.lib linker library in a binary coded decimal format. The version number changes only with major revisions/enhancements. The release number changes with every new upgrade/release of the package. Registered users are offered upgrade options for new versions and releases. The level number changes when small bugs or problems are fixed, or with very minor enhancements. Users are not notified of level changes.

Returns The library version, release, and level are returned in BCD format. The return value is broken down as follows:

`0x00vvrrll`

Where vv is the version number, rr is the release number, and ll is the level number. For instance, version 8.99.1 would be returned as hex 0x00089901.

Function **gs_zuser_suggestions**
User suggestions implemented in GDS

Synopsis `long gs_zuser_suggetions( lots'o'suggestions );`

Description This is not a real function, but it could be. The GameSmith Development system is an extendable system that is constantly under development. Oregon Research is totally

committed to bringing you the very best game development environment possible.

We actively request your suggestions on how to make this an even better tool for you and the Amiga. If you have a function you want to see implemented, then write it down and send it to us. We want to hear from you!

See also

Your imagination.

Addendum

Added new "gs_plot_pixel_fast" routines that are MUCH faster than the previous plot pixel versions. These routines require a pointer to a bitmap offset table, and so two routines are provided to build such a table and to free it from memory.

```
void gs_plot_fast(struct BitMap *,int *bm_offset,int X,int Y,int color);
                        a0          a1          d0    d1    d2

void gs_plot_reverse_fast(struct BitMap *,int *bm_offset,int X,int Y);
                        a0          a1          d0    d1

int gs_plot_test_fast(struct BitMap *,int *bm_offset,int X,int Y);
d0          a0          a1          d0    d1

int *gs_alloc_BMOT(struct BitMap *);
```

Where a nonzero return value is a pointer to the bitmap offset table. A NULL return indicates there was not enough memory to allocate the table.

```
void gs_free_BMOT(int *bm_offset);
```

Added new flag bit ANIMLOAD_LOCKCHECK to the anim_load structure. Setting this bit will cause a load error (new error code -9) if the anim object being loaded is NOT locked. This prevents other people from replacing your anim objects with their own without your program knowing it.

A feature was added to the animation system which yields a 25% - 30% speed increase for objects that save/restore background data, and 10% - 15% over objects that use the "erase" method. By simply supplying a "spare" bitmap which contains a pure background image, the animation system can remove the "save" step altogether. The image blit process then becomes simply blit/restore instead of save/blit/restore, thus producing the significant performance boost. This also has 2 nice side effects if you can find the Chip RAM for the extra bitmap:

- 1) Adding objects to an animation display list becomes MUCH faster.
- 2) Since no object save areas are required, your application will never run into memory problems when trying to add new objects to a display list. Also, if the application is using many save/restore objects in a double buffered display, the amount of extra memory required for the spare bitmap isn't much more than the other method.

The "spare" bitmap must be the exact same size as the target display bitmap(s).

To make use of this new feature, the animation function `gs_init_anim()` now accepts a pointer to another bitmap as the 4th parameter. From assembler, this pointer needs to be loaded into register a2. If your application will not make use of a spare (or secondary) bitmap, then this parameter must be NULL.

For example:

```
gs_init_anim(dlist,bitmap1,bitmap2,bitmap3); /* double buffered with spare */
           d0      a0      a1      a2
or
gs_init_anim(dlist,bitmap1,NULL,bitmap3); /* single buffered with spare */
gs_init_anim(dlist,bitmap1,bitmap2,NULL); /* double buffered no spare */
gs_init_anim(dlist,bitmap1,NULL,NULL); /* single buffered no spare */
```


This is all that's required to make use of this new feature! After a display list is initialized with a spare bitmap pointer, adding objects that save/restore background data will no longer require a save area (the animation system is smart enough not to allocate a save area in the event that a spare "restore" bitmap was supplied during initialization). Note that this method is also more than 10% faster than the "erase" display method, so if you are supplying a spare bitmap, it's best to make sure that the objects are displayed using the save/restore method (see demos "bubble_blank" and "left_right" for good examples of how this is accomplished depending on whether a spare bitmap could be allocated).

To briefly view the demos, click on (or execute from the CLI) either the DEMO or DEMO_AGA scripts in the examples directory.

IMPORTANT! To exit each demo, click on the right mouse button. If the demo does not immediately exit, hold the right mouse button down until it does exit. The very last "demo" executed from the script is a tank battle game. You will need a joystick plugged into port 2 in order to select game options (one of which is the "Quit" option).

The demo scripts do not exercise all demo options/modes. See the respective document files and/or demo source code for more information. All demos were designed to run from the CLI, and many will display parameter information by simply running them without any parameters.

Some of the demos, especially the scroller demos, allow user control via a joystick plugged into port 2.

General Index

Symbols

#define 162
68000 machines 10

A

Acknowledgment 6

Addendum 188

AGA

- Dual Playfield 40
- Mode Promotion 40
- modes 30, 40
- Parallax Viewport 42, 46
- Scrolling 40
- unpromoted modes 37, 39

AmigaDOS Librarian 100

- using Assembler 100
- using 'C' 100

Anim 49

- Cell Size 77
- Child 56
- Collision Table 87
- Colors 77
- Creating 74
- Disk- based 71
 - advantages of 71

Examples of 50

Files 83

- contents of 83
- creating 71
- Final Output 83
- Locked 84

ID 80

Saving 83

Linked 92

Names 74

Options 78

Renaming 80

Saving 83

Selecting 80

- Structure 59, 61
- Tutorial 8

Anim Complexes 14, 51

- Files 84
- Creating 81
- examples of 51
- Renaming 82
- Saving 83
- Structure 65

Anim Header Files 84

Anim Load Structure 66

Anim Source 92

Animation

- Anim Complex Structure 65

- Anim Load Structure 66

- Anim Structures 59 - 68

- Animation Functions 68

- Attached Sprites 68

- Background Collision Structure 59

- Bounds Checking 53

- Collision Detection 53

- Complex Object 51. See also
 - anim complexes

- Display lists 49

- Display Methods 55

- Hardware Sprites 67

- Linked Anim Objects 57

- Loop 50

- Object Arrays 58

- Object Collision Structure 60

- Object Save Areas 51

- Object Types 50

- Real Space 52

- Simple Object 50

- Single shot 50

- Special Effects 55

- Object Flicker 55

- Translucency 56

- Virtual Space 52

Animation Functions 68

Animation Preview 12, 80

Animation System

- CITAS Objects 72

- features of 48

- Structure Hierarchy 49

ANSI 107

Assembler 1, 107

Attached Anims 56, 62

Attached Sprites 68

Audio Filter Control 103

B

Background

- Restoring 51

- Saving 51

Background Collision Structure 59

Bitmap Allocation 104

Bitmap Handling 49

BitMapHeader 22

Bitmaps 19, 22, 29, 33

- Allocation 104
 - Bit mask 21
 - Bounds Checking 53
 - Display Method 55
 - Dual Playfield 58
 - Interleaved 23
 - Limits 23
 - Offsets 34
 - Structure 22
 - Blit Image 79
 - Blitter 23
 - Collision 26
 - CPU 26
 - Mask 24
 - Structure 22
 - Blitter Functions 30
 - Bounds Checking 53, 64
- ## C
- "C" Language 1, 106
 - Chained Objects. See also Attached Anims
 - char 108
 - CITAS 1, 71
 - Anim 74
 - Anim complex 51
 - Anim Options 78
 - Binary Files 1
 - Blitter 23
 - Cell Info 77
 - Collision 88
 - Collision Detection 53
 - Copy 76
 - Copy Color 77
 - Disk Based Anims 57
 - Flip All 76
 - Image Alignment 77, 81
 - Keyboard Controls 95
 - Loading and Sequencing Image Cells 75
 - Locked Anim Files 106
 - Object names 51
 - Orient 76
 - Position Image 76
 - Source Output 92
 - System Preferences 73
 - Translucency 79
 - Tutorial 11
 - Collision Detection 53
 - Area 59, 60
 - Modifying 90
 - Rectangle 89
 - Type 89
 - Collision Handler 15, 53, 54
 - Enable 59, 60, 63, 91
 - Number of 53
 - Object-to-Background 54
 - Bitplanes 15
 - Collision Enable Mask 91
 - Defining 91
 - limitations of 54
 - Tutorial 15
 - Object-to-Object 53
 - Defining 88
 - Tutorial 12, 15
 - Collision Masks 15, 86, 91
 - Collision Points 91
 - modifying 12, 92
 - Collision Tables 13, 54, 86
 - building 86
 - Saving 87
 - Select anim 87
 - size of 54
 - Color palette 10, 42
 - Color Table 19, 20, 23, 33
 - Number of Colors 20, 33
 - Sliced Parallax 44
 - Copper List
 - Commands 27
 - GameSmith Display 28
 - Instructions 27
 - Intuition Display 28
 - Parallax 42
 - Sliced Parallax 43
 - Copper Structure 27
 - Copyright Notifications ii
 - CPU Blittable Objects 6
 - Creating a Simple Anim 74
 - Creating an Anim Complex 81
 - Credits ii
 - CYDEC 101. See also Encryption
- ## D
- Decryption 101
 - Demo programs 7
 - Disk Based Anim Objects 57
 - Display Functions 47, 110
 - Display lists 49
 - priority 49, 50
 - Display Methods 55
 - centering 37
 - Erase Image 55, 78
 - Save/Restore Background 55, 78
 - Simple Copy 55, 79

- Display Modes 36
 - CITAS 72
 - NTSC 10
 - PAL 10
- Display Priority 36, 49, 63, 79
- Display Structure 31, 35
- Display System 31
- Double Buffering 58, 73
 - copper lists 32
 - pages 31
 - technique 31
- Dual Playfield
 - AGA 41
 - Bitmaps 58
 - Objects in 58
 - Priority 37
 - Promoted display 40

E

- Electronic Support
 - CompuServe 5
 - Genie 5
 - Internet 5
- Encryption 83, 100, 101
 - Hints 102
 - Linking 102
 - Picture Loader 19
 - Using files with 106
 - WARNING 102
- Error Messages 4

F

- File loader
 - Anim 23
- File Requestor 103
- Final Output 83
 - Locked 84
- Flags
 - Anim Structure 63
 - Anim Load Structure 67
 - ANIM_ACTIVE 63
 - ANIM_BOUNDS_X1 64
 - ANIM_BOUNDS_X2 64
 - ANIM_BOUNDS_Y1 64
 - ANIM_BOUNDS_Y2 64
 - ANIM_CLEAR 63
 - ANIM_COLLISION 63
 - ANIM_COLLISION_BG 63
 - ANIM_COPY 63
 - ANIM_CPUBLIT 64

- ANIM_FLICKER 63
- ANIM_INVISIBLE 63
- ANIM_MERGE 63
- ANIM_ONSCREEN 63
- ANIM_PARENT 63
- ANIM_REVERSE 63
- ANIM_SAVE_BG 63
- ANIM_VCOLLIDE 64
- ANIM_VSPACE 64
- ANIMLOAD_FASTRAM 67
- ANIMLOAD_LOCKCHECK 189
- ANIMLOAD_NOCOLOR 67
- ANIMLOAD_NOFASTRAM 67
- BLIT_1SHOT_BACKWARD 26
- BLIT_1SHOT_FORWARD 25
- BLIT_CPUBLIT 26
- BLIT_DATACOPY 26
- BLIT_MERGE 25
- Blitter 25
- Display Structure 36
- GSV_BRDRBLNK 37
- GSV_DDFADJUST 37
- GSV_DOUBLE 36
- GSV_FLIP 36
- GSV_HARDX 37
- GSV_HARDY 37
- GSV_PF2PRI 37
- GSV_SCROLL1 37
- GSV_SCROLL2 37
- GSV_SCROLLABLE 37
- GSVP_ALLOCBM 34
- GSVP_DCLIP 34
- GSVP_DPF 34
- GSVP_NOCOLOR 34
- GSVP_NOWAIT 34
- Hard Copper 29
- ILBM_ALLOC1 21
- ILBM_ALLOC2 21
- ILBM_COLOR 21
- JOY_BUTTON1 102
- JOY_BUTTON2 102
- JOY_DOWN 102
- JOY_LEFT 102
- JOY_RIGHT 102
- JOY_UP 102
- loadILBM 21
- NTSC_MONITOR_ID 40
- PAL_MONITOR_ID 40
- Parallax Viewport 43
- PVP_AGACOLOR 43
- PVP_DPF 43
- PVP_DPF1 43

- PVP_DPF2 43
- PVP_LINE1 43
- PVP_RELOAD 43
- SND_EFX 98
- SND_FAST 98
- UC_END 27
- UC_MOVE 27, 29
- UC_PV 29
- UC_PVP 30
- UC_SETCOLOR 27, 29
- UC_SETCOLORAGA 29
- UC_SPVP 29, 30
- UC_WAIT 27, 29
- UCF_DOUBLE 29
- Viewport Structure 34
- Forbid()
- Functions, references in body text,
 - see also Library reference for full descriptions
- Animation
 - gs_draw_anims 53
 - gs_init_anim 58
 - gs_rs_offset 52
- Display
 - gs_alloc_spvp 44
 - gs_create_display 27, 31
 - gs_display 31
 - gs_flip_display 36
 - gs_free_spvp 44
 - gs_remove_display 28, 34
 - gs_scroll_vp 38
 - gs_viewport 28, 42
- Encryption
 - gs_cypher 19, 101, 102
 - gs_cypher_block 101
 - gs_decrypt_data 101
 - gs_decypher 101, 102
 - gs_encrypt_data 101
 - gs_identify_cypher 101
- Graphics
 - gs_free_bitmap 19
 - gs_free_hard_copper 28
 - gs_get_bitmap 19, 20, 34, 35
 - gs_hard_copper 28
 - gs_replace_ucop 28
 - gs_user_copper 27, 28

G

- Game Design 8
- Games 8
 - Defining the Parts 8

- Design 8
- Objects 11
- Sound 9
- GameSmith Development System 1, 2
- Graphics
 - Designing 8, 10
 - System 19
- Graphics Functions 30, 110

H

- Halfbrite 72
- HAM 72
- HAM8 72
- Hard Copper Structure 29
- Hard drive 3
- Hardware Sprites 67
- Header Files
 - Anim Files 84
 - GameSmith.i 28
 - LIBPTRS.I 100
 - MODEID.H 40
 - NOTES.H 98
 - NOTES.I 98
- HiSoft 5

I

- IFF 101
 - 8SVX 96, 101
 - ILBM 19, 71, 101
 - BMHD chunk 22
 - format 22
- Image Creation 71
- Installation 3
- int 108
- Integer size 106

J

- Joystick Polling 102

L

- Librarian Functions 104
- Library Quick Reference 110
- Library Reference
 - Understanding Symbols 108
- Linked Anim Objects 57
- Linking 105
 - BLINK 105
 - SLINK 105

Load Image 75
load_iff_sound 101
loadILBM 19, 101
Locked Anim Files 57, 106
long 108

M

MASKS 13
Mastering GameSmith 6
 Animation 49
 Demo Programs 7
 Tutorial 6, 8
Mode Promotion 40
 preventing 40
Multiple Viewports 38

O

Object Animation 80
Object Arrays 58
Object Collision Structure 60
Object Flicker 55, 63
Object priority 14
 Playfields 37
Object-to-Background Collision 90
Object-to-Object Collision 14, 87
Oregon Research 5
Output Types 71

P

Parallax Viewports 42
 Copper Instruction 42
 limitations 42
 pvp structure 42
 Sliced 44
Parameters
 in registers 107
 on stack 107
Parent Anims 56
Permit() 28
Picture Functions 30
Picture Loader 19
Plot Routines 104
Pseudo-code 16

R

Random Number Generator 103
Real Space 52
 Collision Detection 54

Rectangle Structure 32
Registration number. See Serial number
RLE compression 71

S

SAS/C compiler 103, 105
 6.5 Users Note 108
Scrolling 39, 39
 AGA 39
Security
 Encryption 100
 Final Lock 73, 84
Serial Number 4, 57, 72
Single Shot Animation
 Backward 79
 Forward 79
Software License Agreement iii
Sound Functions 99, 113
Sound system 95
 IFF 8SVX 16, 98
 sampling hardware 16
 Software Interrupt RAM Cache 96, 97
 Sound Structure 97
 Tutorial 16
 User Callable Routines 96
 Vertical Blank Interrupt Controller 96
Source Code Labels 93
 Assembler Symbols 94
 ‘C’ Symbols 93
Source Output 71, 92
Special Effects 55
Sprites 67
 Display Priority 36
struct 108
Structures
 anim_cplx 65
 anim_load_struct 66
 anim_struct 61
 BitMap 22
 BitMapHeader 22
 blit_struct 23
 coll_bg_struct 59
 collision_struct 60
 copper_struct 27
 display_struct 35
 gs_pvp 42
 gs_rectangle 32
 gs_spvp 44
 gs_viewport 32
 hard_copper 29
 loadILBM_struct 19

- RasInfo 35
- RastPort 9
- sound_struct 97
- UCopList 27
- ViewExtra 36
- ViewPortExtra 34

T

- Table of Contents iv
- Task Priority 103
- Technical Support 4
- Translucency 56, 79
- Tutorial 6, 8

U

- unsigned 108
- Using Encrypted Graphics and Sound Files 106
- Using Locked Anim Files 106
- Using the Libraries 105
 - From Assembler 107
 - From 'C' 106
 - Linking 105
- Utility Functions 100, 104, 113

V

- Vector Routines 102
- Vertical Blank Functions 104
- Vertical Blank Timer 104
- Viewport
 - Definition 31
- Viewport Structure
 - Sliced 44
- Viewports
 - clipping 35
 - Multiple 38
 - Parallax 42
 - drawing 46
 - Sliced 44
 - Scrollable 36, 38
 - bits lost 39
 - scroll_type 46
 - size of 32
 - Sliced Structure 44
 - Tutorial 10
- Virtual Space 52
- void 108

GameSmith Library

Index by System

Animation

gs_add_anim 114
gs_add_anim_cplx 115
gs_add_parent 116
gs_anim_backward 118
gs_anim_children 118
gs_anim_cplx 119
gs_anim_cplx_children 118
gs_anim_cplx_parent 120
gs_anim_forward 120
gs_anim_obj 120
gs_anim_parent 121
gs_clear_anim 127
gs_clear_anim_flicker 128
gs_clear_anim_merge 128
gs_clear_collision 128
gs_clear_collision_bg 129
gs_clear_cplx 129
gs_clear_cplx_flicker 130
gs_clear_cplx_merge 130
gs_clear_sprite 130
gs_cplx_backward 131
gs_cplx_forward 131
gs_disable_anim_bounds 133
gs_disable_anim_collision 133
gs_disable_anim_collision_bg 133
gs_disable_cplx_collision 134
gs_disable_cplx_collision_bg 134
gs_draw_anim_objs 136
gs_draw_anims 136
gs_enable_anim 137
gs_enable_anim_bounds 137
gs_enable_anim_collision 137
gs_enable_anim_collision_bg 138
gs_enable_cplx 138
gs_enable_cplx_collision 138
gs_enable_cplx_collision_bg 139
gs_free_anim 141
gs_free_anim_save_area 142
gs_free_bitmap 142
gs_free_cplx 143
gs_free_cplx_save_area 143
gs_free_parent_save_area 144
gs_get_anim_save_area 145
gs_get_cplx_save_area 146
gs_get_parent_save_area 149
gs_init_anim 150
gs_load_anim 152

gs_move_anim 158
gs_move_cplx 158
gs_move_sprite 159
gs_next_anim_page 159
gs_re_dim 165
gs_remove_anim 165
gs_remove_cplx 165
gs_restore_anim_bg 167
gs_reverse_anim 168
gs_reverse_cplx 168
gs_rs_offset 169
gs_rs_window 169
gs_set_anim_bounds 172
gs_set_anim_cell 173
gs_set_anim_flicker 173
gs_set_anim_merge 174
gs_set_collision 174
gs_set_collision_bg 175
gs_set_cplx_cell 176
gs_set_cplx_flicker 176
gs_set_cplx_merge 177
gs_set_cplx_prio 176
gs_set_cplx_seq 177
gs_sort_anims 180

Display

gs_alloc_BMOT 188
gs_alloc_spvp 118
gs_create_display 132
gs_display 134
gs_flip_display 141
gs_free_display_list 143
gs_free_BMOT 188
gs_free_spvp 144
gs_get_display_list 147
gs_LoadRGB 158
gs_old_display 160
gs_remove_display 166
gs_scan_copper 169
gs_scroll_vp 170
gs_scroll_vp_pf1 171
gs_scroll_vp_pf2 171
gs_SetRGB 179
gs_show_display 179

Graphics

gs_blit_clear 121
gs_blit_copy 122
gs_blit_copy_fine 122
gs_blit_copy2 123
gs_blit_fill 123
gs_blit_free_save_area 124

gs_blit_get_save_area 124
gs_blit_image 124
gs_blit_image_fine 125
gs_blit_image2 126
gs_blit_memcpy 126
gs_blit_restore 127
gs_blit_save_bg 127
gs_chiprev 127
gs_free_blitter 142
gs_free_hard_copper 144
gs_get_bitmap 145
gs_get_blitter 146
gs_hard_copper 150
gs_plot 162
gs_plot_fast 188
gs_plot_reverse 163
gs_plot_reverse_fast 188
gs_plot_test 164
gs_plot_test_fast 188
gs_replace_ucop 167

Librarian

gs_close_libs 130
gs_open_libs 160

Picture

gs_get_ILBM_bm 148
gs_loadILBM 157

Sound

gs_close_sound 131
gs_free_sound 144
gs_load_iff_sound 155
gs_load_raw_sound 156
gs_open_sound 161
gs_set_period 178
gs_set_volume 178
gs_start_sound 181
gs_stop_sound 182
gs_sound_check 180

Utility

gs_add_vb_server 117
gs_file_requestor 139
gs_joystick 151
gs_LEDoff 152
gs_LEDon 152
gs_random 164
gs_remove_vb_server 167
gs_task_prio 182
gs_user_copper 182
gs_version 187

gs_init_vector 151
gs_step_vector 181
gs_ustep_vector 185
gs_close_vb_timer 131
gs_open_vb_timer 162
gs_vb_time 186
gs_vb_timer_reset 187
gs_zuser_suggestions 187

GameSmith Library

Index – Alphabetical

A

- gs_add_anim 114
- gs_add_anim_cplx 115
- gs_add_parent 116
- gs_add_sprite 116
- gs_add_vb_server 117
- gs_alloc_BMOT 188
- gs_alloc_spvp 118
- gs_anim_backward 118
- gs_anim_children 118
- gs_anim_cplx 119
- gs_anim_cplx_children 119
- gs_anim_cplx_parent 120
- gs_anim_forward 120
- gs_anim_obj 120
- gs_anim_parent 121

B

- gs_blit_clear 121
- gs_blit_copy 122
- gs_blit_copy_fine 122
- gs_blit_copy2 123
- gs_blit_fill 123
- gs_blit_free_save_area 124
- gs_blit_get_save_area 124
- gs_blit_image 124
- gs_blit_image_fine 125
- gs_blit_image2 126
- gs_blit_memcpy 126
- gs_blit_restore 127
- gs_blit_save_bg 127

C

- gs_chiprev 127
- gs_clear_anim 127
- gs_clear_anim_flicker 128
- gs_clear_anim_merge 128
- gs_clear_collision 128
- gs_clear_collision_bg 129
- gs_clear_cplx 129
- gs_clear_cplx_flicker 130
- gs_clear_cplx_merge 130
- gs_clear_sprite 130
- gs_close_libs 130
- gs_close_sound 131
- gs_close_vb_timer 131
- gs_cplx_backward 131
- gs_cplx_forward 217

- gs_create_display 218

D

- gs_disable_anim_bounds 133
- gs_disable_anim_collision 133
- gs_disable_anim_collision_bg 133
- gs_disable_cplx_collision 134
- gs_disable_cplx_collision_bg 134
- gs_display 134
- gs_draw_anim_objs 136
- gs_draw_anims 136

E

- gs_enable_anim 137
- gs_enable_anim_bounds 137
- gs_enable_anim_collision 137
- gs_enable_anim_collision_bg 138
- gs_enable_cplx 138
- gs_enable_cplx_collision 138
- gs_enable_cplx_collision_bg 139

F

- gs_file_requestor 139
- gs_flip_display 141
- gs_free_anim 141
- gs_free_anim_save_area 142
- gs_free_bitmap 142
- gs_free_blitter 142
- gs_free_BMOT 188
- gs_free_cplx 143
- gs_free_cplx_save_area 143
- gs_free_display_list 143
- gs_free_hard_copper 144
- gs_free_parent_save_area 144
- gs_free_sound 144
- gs_free_spvp 144

G

- gs_get_anim_save_area 145
- gs_get_bitmap 145
- gs_get_blitter 146
- gs_get_cplx_save_area 146
- gs_get_display_list 147
- gs_get_ILBM_bm 148
- gs_get_parent_save_area 149

H

- gs_hard_copper 150

I

- gs_init_anim 150
- gs_init_vector 151

J

gs_joystick 151

L

gs_LEDoff 152
gs_LEDon 152
gs_load_anim 152
gs_load_iff_sound 155
gs_load_raw_sound 156
gs_loadILBM 157
gs_LoadRGB 158

M

gs_move_anim 158
gs_move_cplx 158
gs_move_sprite 159

N

gs_next_anim_page 159

O

gs_old_display 160
gs_open_libs 160
gs_open_sound 161
gs_open_vb_timer 162

P

gs_plot 162
gs_plot_fast 188
gs_plot_reverse 163
gs_plot_reverse_fast 188
gs_plot_test 164
gs_plot_test_fast 188

R

gs_random 164
gs_re_dim 165
gs_remove_anim 165
gs_remove_cplx 165
gs_remove_display 166
gs_remove_vb_server 167
gs_replace_ucop 167
gs_restore_anim_bg 167
gs_reverse_anim 168
gs_reverse_cplx 168
gs_rs_offset 169
gs_rs_window 169

S

gs_scan_copper 169
gs_scroll_vp 170
gs_scroll_vp_pf1 171

gs_scroll_vp_pf2 171
gs_set_anim_bounds 172
gs_set_anim_cell 173
gs_set_anim_flicker 173
gs_set_anim_merge 174
gs_set_collision 174
gs_set_collision_bg 175
gs_set_cplx_cell 176
gs_set_cplx_flicker 176
gs_set_cplx_merge 177
gs_set_cplx_prio 176
gs_set_cplx_seq 177
gs_set_period 178
gs_set_volume 178
gs_SetRGB 179
gs_show_display 179
gs_sort_anims 180
gs_sound_check 180
gs_start_sound 181
gs_step_vector 181
gs_stop_sound 182

T

gs_task_prio 182

U

gs_user_copper 182
gs_ustep_vector 185

V

gs_vb_time 186
gs_vb_timer_reset 187
gs_version 187

Z

gs_zuser_suggestions 188